



Christopher
Martel

Visual C# 2010

Praxiseinstieg

Inhaltsverzeichnis

A	Windows Forms	5
A.1	Grundaufbau einer Windows-Forms-Anwendung	6
A.1.1	Die Klasse Application	8
A.1.2	Ein Formular erzeugen	10
A.1.3	Designer vs. Code	19
A.1.4	Für Ereignisse registrieren	21
A.2	Allgemeine Steuerelemente	23
A.2.1	Button	24
A.2.2	CheckBox	26
A.2.3	CheckedListBox	28
A.2.4	ComboBox	30
A.2.5	DateTimePicker	31
A.2.6	Label	32
A.2.7	LinkLabel	33
A.2.8	ListBox	34
A.2.9	ListView	36
A.2.10	MaskedTextBox	37
A.2.11	MonthCalendar	39
A.2.12	NumericUpDown	39
A.2.13	PictureBox	40
A.2.14	ProgressBar	41
A.2.15	RadioButton	42
A.2.16	TextBox	42
A.2.17	RichTextBox	43
A.2.18	TreeView	44
A.3	Menüs und Statusleisten	47
A.3.1	MenuStrip	47
A.3.2	StatusStrip	48
A.3.3	ToolStrip	48
A.3.4	ToolStripContainer	49
A.3.5	ContextMenuStrip	49
A.4	Layout und Container	50

A.4.1	Die Eigenschaften Dock und Anchor	50
A.4.2	FlowLayoutPanel	52
A.4.3	GroupBox	52
A.4.4	Panel	53
A.4.5	SplitContainer	53
A.4.6	TabControl	54
A.4.7	TableLayoutPanel	56
A.5	Allgemeine Dialoge	57
A.5.1	ColorDialog	57
A.5.2	FontDialog	58
A.5.3	FolderBrowserDialog	59
A.6	Drucken	60
A.6.1	PrintDocument	60
A.6.2	PrintDialog	62
A.6.3	PageSetupDialog	63
A.6.4	PrintPreviewDialog	64
A.6.5	PrintPreviewControl	65
A.7	Benutzerdefinierte Steuerelemente	66
A.8	Einstellungen und Ressourcen	68
A.9	Zusammenfassung	70
B	Grafikprogrammierung mit GDI+	71
B.1	Was ist GDI+?	71
B.1.1	Namensräume und Klassen für die Grafikprogrammierung	72
B.2	Das Koordinatensystem	72
B.2.1	Die Klasse Graphics	73
B.2.2	Transformation von Objekten	74
B.3	Mit GDI+ zeichnen	77
B.3.1	Linien	77
B.3.2	Polyline	78
B.3.3	Polygone	79
B.3.4	Rechtecke	81
B.3.5	Splines	82
B.3.6	Bézierkurven	84
B.3.7	Ellipsen und Kreise	85
B.3.8	Segmente	86
B.3.9	Bogen	87
B.3.10	Pfad	88

B.4	Mit Bildern arbeiten	91
B.4.1	Das Steuerelement PictureBox.	91
B.4.2	Die Klasse Image.	91
B.4.3	Eigenschaften und Methoden der Klasse Image	92
B.4.4	Transformation von Bildern	95
B.4.5	Punkte ermitteln und setzen	95
B.5	Textausgabe	96
B.5.1	Fonts	97
B.6	Farbangaben	98
B.6.1	Die Struktur Color.	98
B.6.2	Das ARGB-Farbsystem.	98
B.7	Zeichnen mittels Pinsel und Stift	99
B.7.1	Die Klasse Pen	99
B.7.2	Die Klasse Brush	102
B.7.3	SolidBrush.	102
B.7.4	LinearGradientBrush	102
B.7.5	TextureBrush	103
B.7.6	PathGradientBrush	104
B.7.7	HatchBrush.	105
B.8	Zusammenfassung	106
	Stichwortverzeichnis	107

Windows Forms

Windows Forms ist die am weitesten verbreitete Technologie zur Erzeugung von grafischen Oberflächen und der Vorläufer von WPF. Entsprechend dem Aussehen ähneln Windows-Forms-Anwendungen klassischen Win32-Anwendungen und werden heutzutage oft noch verwendet. Besonders in mobilen Systemen, die das Betriebssystem Windows Mobile bzw. Windows CE nutzen, kommt diese Technologie innerhalb des .NET Compact Frameworks zum Einsatz.

In diesem Kapitel werden wir uns mit der Erzeugung von Anwendungen mit grafischen Oberflächen mithilfe der Technologie Windows Forms beschäftigen. Sie werden dabei Folgendes lernen:

- Grundaufbau einer Windows-Forms-Anwendung
- Verwendung allgemeiner Steuerelemente wie Buttons und Labels
- Erzeugung von Menüs
- Festlegung von dynamischen Layouts zur Platzierung von Steuerelementen
- Verwendung allgemeiner Dialoge, wie zum Beispiel zur Anzeige von Farbeinstellungen
- Drucken von Informationen aus einer Windows-Forms-Anwendung
- Erzeugung von benutzerdefinierten Steuerelementen

Hinweis

Windows Forms stellt eine Technologie dar, die mit C# genutzt werden kann. Sie ist jedoch keine C#-spezifische Technologie, sondern kann auch im Zusammenhang mit anderen .NET-Sprachen, wie beispielsweise VB.NET, verwendet werden.

Hinweis

In diesem Kapitel wird auf ein großes Beispiel verzichtet, da es mit Windows Forms und dem Designer aus Visual Studio noch einfacher als in WPF ist, grafische Oberflächen umzusetzen. Im Gegensatz zu WPF wird in Windows Forms viel mit dem Designer gearbeitet. Eine strikte Trennung von Geschäftslogik und grafischen Komponenten ist jedoch nicht so einfach möglich wie in WPF.

Die einzelnen Beispiele in diesem Kapitel geben Ihnen eine gute Übersicht darüber, wie Sie leicht eigene Anwendungen in Windows Forms erzeugen können.

Hinweis

Sie werden in diesem Kapitel die Grundlagen der Oberflächenprogrammierung mit Windows Forms kennenlernen. Danach werden Sie in der Lage sein, eigene Windows-Forms-Anwendungen zu erzeugen. Wenn Sie jedoch weitergehende Informationen zu dem Thema benötigen, sei an dieser Stelle auf entsprechende Fachliteratur verwiesen.

A.1 Grundaufbau einer Windows-Forms-Anwendung

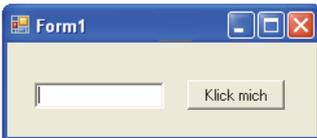
Jede Anwendung hat einen Einstiegspunkt. Dies ist wie bei Konsolenanwendungen die Klasse **Program** mit der Methode **Main**. Wenn Sie zur Codeansicht der Klasse wechseln, werden Sie den folgenden Aufbau der Methode **Main** sehen:

```
static class Program
{
    ///<summary>
    ///Der Haupteinstiegspunkt für die Anwendung
    ///</summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

Listing A.1: Definition der Klasse Program innerhalb einer Windows-Forms-Anwendung

Wie Sie hier erkennen können, ist die Methode **Main** mit dem Attribut **[STAThread]** markiert. Mithilfe dieser Angabe wird die Methode COM-kompatibel gemacht. Generell verwendet Windows Forms kein COM, allerdings wird diese Interoperabilität zur Kommunikation mit dem Betriebssystem von einigen Komponenten in Windows Forms benötigt. Weiterhin ist COM erforderlich, um über Nachrichten im Sinne von Events, die auf Betriebssystemebene ausgetauscht werden, zwischen den einzelnen Komponenten zu kommunizieren.

Der Aufruf der Methode **EnableVisualStyles** ermöglicht es, die Steuerelemente dem Layout von Windows XP anzupassen. Die folgende Abbildung zeigt den Unterschied zwischen dem Standardaussehen und dem Aussehen von Windows XP bzw. den höheren Betriebssystemversionen wie Windows Vista und Windows 7:



Ohne EnableVisualStyles



Mit EnableVisualStyles

Abb. A.1: Unterschied mit und ohne EnableVisualStyles

Die Methode **SetCompatibleTextRenderingDefault** erwartet einen booleschen Parameter. Wird ihr **true** übergeben, wird für die Textausgabe der Steuerelemente GDI verwendet, ansonsten GDI+.

Hinweis

GDI und GDI+ wird in einem separaten Kapitel, das Sie kostenlos von der Verlagsseite herunterladen können, behandelt.

Zu guter Letzt wird der Methode **Run** die Instanz der Formularklasse angegeben, die als Hauptformular angezeigt wird. Die Methode startet im Hintergrund eine Nachrichtenschleife, mit der Maus- und Tastatureingaben, aber auch Nachrichten zwischen den einzelnen Komponenten und Anwendungen verarbeitet werden können. Ohne den Aufruf dieser Methode können keine Ereignisse in Windows-Forms-Anwendungen verarbeitet werden. Standardmäßig wird dem Projekt beim Erzeugen einer Windows-Forms-Anwendung ein Formular mit der Bezeichnung **Form1** hinzugefügt. Sie können der Methode **Run** jedoch auch eine andere Instanz einer von der Klasse **Form** abgeleiteten Klasse übergeben, die beim Start als Hauptformular dienen soll.

Jedes Formular hat folgenden grundlegenden Aufbau:

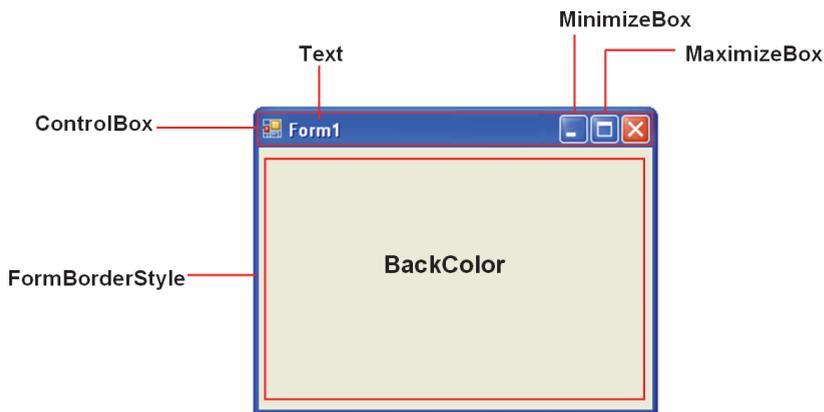


Abb. A.2: Grundaufbau eines Formulars

Die entsprechende Codedarstellung sieht folgendermaßen aus:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    ...//Weitere Definitionen
}
```

Listing A.2: Die Grundstruktur der Formularklasse

Die Methode **InitializeComponent** initialisiert das Formular über die dem Formular per Designer zugewiesenen Steuerelemente und Eigenschaften. Diese sind in einer separaten Datei **FORM1.DESIGNER.CS** definiert, die Sie in Abschnitt A.1.2 kennenlernen werden. Zunächst soll hier aber die Klasse **Application** betrachtet werden, die das Zentrum einer Applikation darstellt.

A.1.1 Die Klasse Application

Die Klasse **Application** repräsentiert die Anwendung an sich und stellt einige statische Methoden, Eigenschaften und Ereignisse zur Verfügung, mit denen Einfluss auf die Anwendung genommen werden kann. Die wichtigsten Eigenschaften der Klasse **Application** sind in den folgenden Tabellen dargestellt:

Pfad der Anwendung und der Anwendungsdaten

Eigenschaft	Bedeutung
string CommonAppDataPath	Liefert den Pfad zu den Anwendungsdaten.
string ExecutablePath	Liefert den Pfad zur ausführbaren Datei der Anwendung (.EXE) inklusive dem Namen der ausführbaren Datei zurück.
string LocalUserAppDataPath	Liefert den Pfad der Anwendungsdaten für den lokalen Benutzer zurück.
string StartupPath	Liefert den Pfad zu ausführbaren Datei der Anwendung ohne den Dateinamen zurück.
string UserAppDataPath	Liefert den Pfad zu den Anwendungsdaten des Anwenders.

Tabelle A.1: Die wichtigsten Eigenschaften zur Ermittlung des Pfades der Anwendung bzw. der Anwendungsdaten mithilfe der Klasse `Application`

Produktdaten ermitteln

Eigenschaft	Bedeutung
string CompanyName	Liefert den Firmennamen, der mit der Anwendung verknüpft ist.
CultureInfo CurrentCulture	Legt die aktuelle Kulturinformation fest bzw. liefert diese zurück.
InputLanguage CurrentInputLanguage	Legt die aktuelle Eingabesprache fest bzw. liefert diese zurück.
string ProductName	Liefert den Produktnamen zurück.
string ProductVersion	Liefert die aktuelle Produktversion zurück.

Tabelle A.2: Die wichtigsten Eigenschaften zur Ermittlung der Produktdaten mithilfe der Klasse `Application`

Registry-Einträge für die Anwendung festlegen

Eigenschaft	Bedeutung
RegistryKey CommonAppDataRegistry	Liefert den gemeinsamen Registry-Eintrag.
RegistryKey UserAppDataRegistry	Liefert den Registry-Eintrag für die Anwendungsdaten zurück.

Tabelle A.3: Die wichtigsten Eigenschaften zur Festlegung von Registry-Einträgen mithilfe der Klasse `Application`

Nachrichtenschleife und offene Formulare

Eigenschaft	Bedeutung
<code>bool AllowQuit</code>	Liefert einen Wahrheitswert (<code>true/false</code>), der angibt, ob die Anwendung beendet werden kann.
<code>bool MessageLoop</code>	Liefert einen Wahrheitswert (<code>true/false</code>), der angibt, ob eine Nachrichtenschleife existiert.
<code>FormCollection OpenForms</code>	Liefert eine Aufzählung von geöffneten Formularen zurück.

Tabelle A.4: Die wichtigsten Eigenschaften zur Ermittlung von Einstellungen zur Nachrichtenschleife und offenen Formularen mithilfe der Klasse `Application`

Die nun folgenden Tabellen zeigen die wichtigsten Methoden und Ereignisse der Klasse `Application`:

Starten, Neustarten und Beenden einer Anwendung

Methode/Ereignis	Bedeutung
<code>public static void Exit()</code>	Beendet die Anwendung.
<code>public static void ExitThread()</code>	Beendet den aktuellen Thread.
<code>public static void Run(Form main-Form)</code>	Öffnet das Hauptformular der Anwendung und startet die Nachrichtenschleife.
<code>public static void Restart()</code>	Beendet die aktuelle Anwendung und startet diese neu.
<code>ApplicationExit</code>	Dieses Ereignis tritt ein, wenn die Anwendung beendet wird.

Tabelle A.5: Die wichtigsten Methoden und Ereignisse zum Starten, Neustarten und Beenden einer Anwendung mithilfe der Klasse `Application`

Zustände und Ausnahmebehandlung

Methode/Ereignis	Bedeutung
<code>public static bool SetSuspendState(PowerState state, bool force, bool disableWakeEvent)</code>	Versetzt das System in den Ruhezustand bzw. in den Standbymodus. Der zurückgelieferte Wahrheitswert gibt an, ob die Aktion erfolgreich war. Die Enumeration <code>PowerState</code> definiert die folgenden Members: <ul style="list-style-type: none"> ■ Suspend: Definiert den Standbymodus. ■ Hibernate: Definiert den Ruhezustand.

Tabelle A.6: Die wichtigsten Methoden und Ereignisse für die Zustandsverwaltung und die Ausnahmebehandlung mithilfe der Klasse `Application`

Methode/Ereignis	Bedeutung
<code>public static void SetUnhandledExceptionMode(UnhandledExceptionMode mode)</code>	Legt fest, wie unbehandelte Ausnahmen behandelt werden sollen. Die Enumeration UnhandledExceptionMode definiert die folgenden Members: <ul style="list-style-type: none"> ■ Automatic: Alle Ausnahmen werden zum <code>ThreadExceptionHandler</code>-Handler umgeleitet, es sei denn, die Konfigurationsdatei der Anwendung spezifiziert einen anderen Handler. ■ ThrowException: Ausnahmen werden nicht zum <code>ThreadExceptionHandler</code>-Handler umgeleitet. ■ CatchException: Alle Ausnahmen werden immer zum <code>ThreadExceptionHandler</code>-Handler umgeleitet.
EnterThreadModal	Dieses Ereignis tritt ein, wenn die Anwendung in den modalen Zustand eintritt.
Idle	Dieses Ereignis tritt ein, wenn die Applikation nichts tut.
LeaveThreadModal	Dieses Ereignis tritt ein, wenn die Anwendung den modalen Zustand verlässt.
ThreadException	Dieses Ereignis tritt ein, wenn eine unbehandelte Ausnahme auf einem Thread ausgelöst wird.

Tabelle A.6: Die wichtigsten Methoden und Ereignisse für die Zustandsverwaltung und die Ausnahmebehandlung mithilfe der Klasse `Application` (Forts.)

Verarbeitung von Ereignissen und Festlegung der visuellen Darstellung

Methode/Ereignis	Bedeutung
<code>public static void DoEvents()</code>	Gibt die CPU zur Abarbeitung weiterer ausstehender Windows-Nachrichten frei.
<code>public static void EnableVisualStyles()</code>	Ermöglicht der Anwendung die Darstellung mithilfe von Styles.

Tabelle A.7: Übersicht über die wichtigsten Methoden und Ereignisse für die Verarbeitung von Ereignissen und die Festlegung der visuellen Darstellung der Klasse `Application`

A.1.2 Ein Formular erzeugen

Ein Formular ist eine Klasse, die von der Basisklasse **Form** abgeleitet ist und sich aus zwei Dateien zusammensetzt: dem Formular mit der Codebehind-Datei und der Datei

FORM1.DESIGNER.CS. Wenn Sie dem Formular beispielsweise ein Steuerelement des Typs BUTTON hinzugefügt haben, sieht der Code der Datei FORM1.DESIGNER.CS ungefähr so aus:

```
partial class Form1
{
    /// <summary>
    /// Erforderliche Designer-Variablen
    /// </summary>
    private System.ComponentModel.IContainer components = null;
    /// <summary>
    /// Verwendete Ressourcen bereinigen,
    /// </summary>
    /// <param name="disposing">True,
    /// wenn verwaltete Ressourcen gelöscht werden sollen;
    /// andernfalls False.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }
    #region Vom Windows Form-Designer generierter Code
    /// <summary>
    /// Erforderliche Methode für die Designer-Unterstützung.
    /// Der Inhalt der Methode darf nicht mit dem Code-Editor
    /// geändert werden.
    /// </summary>
    private void InitializeComponent()
    {
        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(38, 37);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(75, 23);
        this.button1.TabIndex = 0;
        this.button1.Text = "button1";
        this.button1.UseVisualStyleBackColor = true;
        //
        // Form1
        //
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(292, 194);
        this.Controls.Add(this.button1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }
    #endregion
    private System.Windows.Forms.Button button1;
}
```

Listing A.3: Inhalt der Datei FORM1.DESIGNER.CS

Diese Aufteilung ist deshalb so gewählt, weil der logische Code vom Code für die Darstellung getrennt werden soll. Die abgeleitete Klasse `Form1` ist also standardmäßig als **partial** markiert. Wenn Sie nun ein neues Formular erzeugen wollen, können Sie mittels Rechtsklick auf das Projekt und dem Menüpunkt HINZUFÜGEN ein neues Formular hinzufügen:

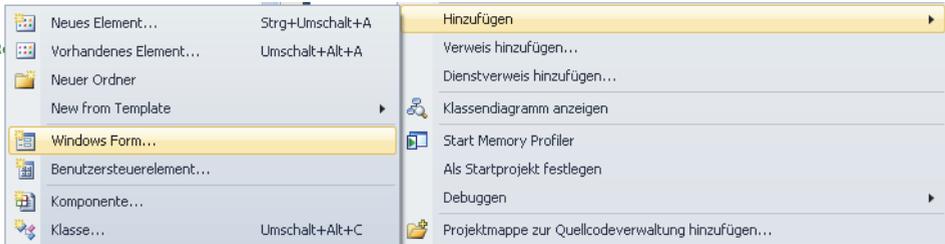


Abb. A.3: Hinzufügen eines neuen Formulars in Visual Studio

Nachdem Sie das Formular mithilfe von Visual Studio erzeugt haben, können Sie die wichtigsten Eigenschaften festlegen. Wie Sie Eigenschaften mithilfe des EIGENSCHAFTENFENSTERS bestimmen können, haben Sie bereits in Kapitel 2 kennengelernt. Die Eigenschaften eines Formulars bzw. eines Steuerelements stellen Eigenschaften einer Klasse dar. Die folgenden Tabellen zeigen die wichtigsten Eigenschaften des Formulars:

Farbe für Schrift und Hintergrund festlegen

Eigenschaft	Bedeutung
<code>BackColor</code>	Legt die Hintergrundfarbe des Formulars fest.
<code>BackgroundImage</code>	Legt das Bild fest, das als Hintergrund verwendet werden soll.
<code>BackgroundImageLayout</code>	Legt das Layout für das Hintergrundbild fest. Es handelt sich um die Enumeration ImageLayout , die folgende Members definiert: <ul style="list-style-type: none">■ Tile: Das Bild wird gekachelt.■ Center: Das Bild wird zentriert.■ Stretch: Das Bild wird gestreckt.■ Zoom: Das Bild wird vergrößert.
<code>ForeColor</code>	Legt die Schriftfarbe fest.

Tabelle A.8: Die wichtigsten Eigenschaften zur Festlegung von Farben für die Schrift und den Hintergrund eines Formulars

Schriftart, Layout und Cursor des Formulars festlegen

Eigenschaft	Bedeutung
Cursor	Legt den Cursor fest, der verwendet werden soll. Es kann aus einer Reihe von verschiedenen Cursorsn ausgewählt werden.
Font	Legt die zu verwendende Schriftart und -größe fest.
FormBorderStyle	Legt die Darstellung des Rahmens und der Titelleiste des Formulars fest.
RightToLeft	Legt fest, ob Text von rechts nach links oder umgekehrt gezeichnet werden soll.
RightToLeftLayout	Legt fest, ob das Layout von rechts nach links angeordnet ist, wenn die Eigenschaft <code>RightToLeft</code> auf <code>Yes</code> gesetzt ist.
IsMdiContainer	Legt fest, ob das aktuelle Formular ein MDI-Container ist.
Opacity	Legt die Transparenz des Formulars in Prozent fest.

Tabelle A.9: Die wichtigsten Eigenschaften zur Festlegung der Schriftart, des Layouts und des Cursors eines Formulars

Bestandteile eines Formulars ändern

Eigenschaft	Bedeutung
ControlBox	Legt fest, ob das Formular über die Schaltflächen zum Minimieren, Maximieren und Schließen verfügt.
HelpButton	Legt fest, ob das Formular in der Titelleiste eine Schaltfläche HILFE besitzt.
Icon	Legt fest, welches Symbol für das Formular verwendet werden soll. Dieses erscheint zusätzlich in der Taskleiste.
MainMenuGrip	Legt das primäre <code>MenuStrip</code> -Element für das Formular fest.
MaximizeBox	Legt fest, ob das Fenster über das Steuerelement MAXIMIEREN innerhalb der Titelleiste verfügt.
MinimizeBox	Legt fest, ob das Fenster über das Steuerelement MINIMIEREN innerhalb der Titelleiste verfügt.
AcceptButton	Legt die Standardschaltfläche ANNEHMEN des Formulars fest. Dadurch kann der Anwender die <code>[Eingabetaste]</code> drücken, bei der ein Klick auf die Schaltfläche »simuliert« wird.
CancelButton	Legt die Standardschaltfläche ABBRECHEN des Formulars fest. Dadurch kann der Anwender die Taste <code>[Esc]</code> drücken, bei der ein Klick auf die Schaltfläche »simuliert« wird.

Tabelle A.10: Die wichtigsten Eigenschaften zur Festlegung der Bestandteile eines Formulars

Größe eines Formulars festlegen

Eigenschaft	Bedeutung
<code>AutoScaleMode</code>	Legt fest, wie die Größe des Formulars angepasst wird, wenn sich die Bildschirmauflösung oder die Schriftart ändert. Der Standard ist Font und legt die Anpassung anhand der Schriftart fest. Weitere Werte sind Dpi für die Bildschirmauflösung sowie None und Inherit .
<code>AutoScroll</code>	Legt fest, ob Laufleisten angezeigt werden sollen, wenn der Inhalt des Formulars größer als das Formular selbst ist.
<code>AutoSize</code>	Legt fest, ob die Größe des Formulars an dessen Inhalt angepasst werden soll.
<code>AutoSizeMode</code>	Legt den Modus fest, in dem die Größe des Formulars durch dessen Inhalt angepasst werden soll. Der Standard ist GrowOnly . Als weiterer Wert kann GrowAndShrink festgelegt werden.
<code>MaximumSize</code>	Legt die Maximalgröße des Formulars in Breite und Höhe fest.
<code>MinimumSize</code>	Legt die Minimalgröße des Formulars in Breite und Höhe fest.
<code>Padding</code>	Legt die Abstände des Inhalts zum Rand des Formulars fest. Die Angabe erfolgt separat für links, oben, rechts und unten.
<code>Size</code>	Legt die aktuelle Größe des Formulars in Breite und Höhe fest.
<code>StartPosition</code>	Legt die Startposition des Formulars fest. Der Wert ist ein Member der Enumeration FormStartPosition , die folgendermaßen definiert ist: <ul style="list-style-type: none">■ CenterParent: Das Formular wird im Zentrum des übergeordneten Formulars angezeigt.■ CenterScreen: Das Formular wird in der Mitte des Bildschirms angezeigt.■ Manual: Das Formular wird entsprechend der für die Eigenschaft <code>Location</code> festgelegten Werte angezeigt.■ WindowsDefaultBounds: Das Formular wird anhand der Standardposition des Betriebssystems und innerhalb des durch das Betriebssystem festgelegten Bereichs angezeigt.■ WindowsDefaultLocation: Das Formular wird an der Standardposition des Betriebssystems angezeigt. Dies ist der Standard.

Tabelle A.11: Die wichtigsten Eigenschaften zur Festlegung der Größe und Position eines Formulars

Eigenschaft	Bedeutung
Location	Legt die Position des Fensters des Formulars in Relation zum Bildschirm fest. Die Angabe erfolgt über eine Instanz der Klasse <code>Point</code> , der sowohl die X- als auch die Y-Koordinate übergeben wird. Der Startpunkt ist dabei die linke obere Ecke des Bildschirms.
Locked	Legt fest, ob das Formular verschoben oder dessen Größe verändert werden kann.
TopMost	Legt fest, ob das aktuelle Formular immer über allen anderen Formularen angezeigt werden soll.

Tabelle A.11: Die wichtigsten Eigenschaften zur Festlegung der Größe und Position eines Formulars (Forts.)

Titelleiste, Bezeichnung und Zustand des Formulars festlegen

Eigenschaft	Bedeutung
Text	Legt den in der Titelleiste anzuzeigenden Text des Formulars fest.
Name	Legt den Instanznamen der Klasse fest. Über diesen Namen kann das Formular im Code verwendet werden.
WindowState	Legt den Zustand des Formulars fest. Der Wert ist ein Member der Enumeration <code>FormWindowState</code> , die folgendermaßen definiert ist: <ul style="list-style-type: none"> ■ Normal: Standardgröße ■ Maximized: Maximiertes Fenster ■ Minimized: Minimiertes Fenster
Enabled	Legt fest, ob das Formular aktiviert ist.
ShowIcon	Legt fest, ob ein Symbol in der Titelleiste angezeigt werden soll.
ShowInTaskbar	Legt fest, ob das Formular in der Taskleiste des Betriebssystems angezeigt werden soll.

Tabelle A.12: Die wichtigsten Eigenschaften zur Festlegung der Titelleiste, der Bezeichnung und des Zustands des Formulars

Sprache und Validierung eines Formulars festlegen

Eigenschaft	Bedeutung
Language	Legt die lokalisierbare Sprache fest.
CausesValidation	Legt fest, ob das Formular Validierungsereignisse auslöst.
AllowDrop	Legt fest, ob auf dem Formular Daten abgelegt werden können, die durch den Benutzer auf das Formular gezogen werden.

Tabelle A.13: Die wichtigsten Eigenschaften zur Festlegung der Sprache und der Validierung eines Formulars

Die folgende Abbildung zeigt die Unterschiede bei der Verwendung der Eigenschaft **FormBorderStyle**:

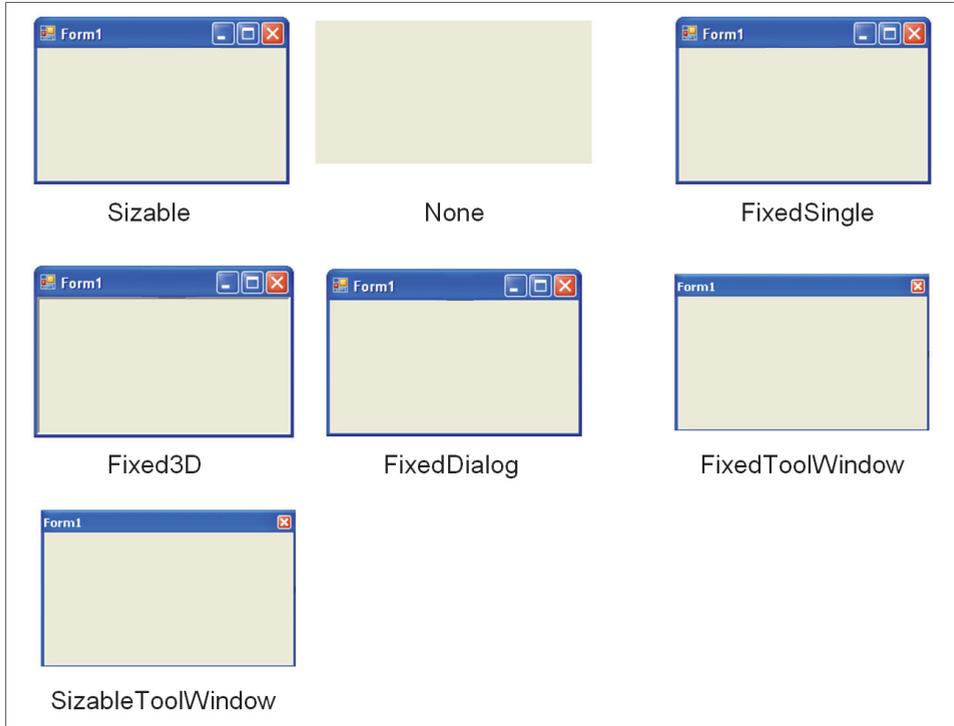


Abb. A.4: Verschiedene Darstellungen für die Werte der Eigenschaft **FormBorderStyle**

Weiterhin verfügt die Klasse **Form** über die folgenden Methoden:

Methode	Bedeutung
<code>public void Activate()</code>	Aktiviert das Formular.
<code>public void BringToFront()</code>	Bringt das Formular in den Vordergrund.
<code>public void Close()</code>	Schließt das Fenster.
<code>public bool Contains(Control ctl)</code>	Liefert einen Wahrheitswert (<code>true/false</code>), der angibt, ob das aktuelle Formular das angegebene Steuerelement beinhaltet.
<code>public void CreateControl()</code>	Erzeugt ein Steuerelement.
<code>public Graphics CreateGraphics()</code>	Liefert eine Instanz der Klasse <code>Graphics</code> zurück (siehe dazu Kapitel GDI+).

Tabelle A.14: Übersicht über die wichtigsten Methoden der Klasse **Form**

Methode	Bedeutung
<pre>public DragDropEffects DoDragDrop(Object data, Drag- DropEffects allowedEffects)</pre>	<p>Startet einen Drag & Drop-Prozess. Dabei definiert data die zu ziehenden Daten und allowedEffects die erlaubten Drag & Drop-Einstellungen. DragDropEffects stellt eine Enumeration dar, die folgende Members definiert:</p> <ul style="list-style-type: none"> ■ All: Die Daten werden kopiert und aus der Quelle entfernt. ■ Copy: Die Daten werden in das Ziel kopiert. ■ Link: Die Daten aus der Quelle werden mit der Ablage verknüpft. ■ Move: Die Daten aus der Quelle werden in das Ziel verschoben. ■ None: Die Daten werden vom Ziel nicht akzeptiert. ■ Scroll: Es wird ein Bildlauf im Ziel durchgeführt. <p>Um mehrere Werte gleichzeitig zu verwenden, können Sie diese durch ein bitweises Oder verknüpft angeben.</p>
<pre>public bool Focus()</pre>	Setzt den Fokus auf das Formular.
<pre>public Control GetNextControl (Control ctl, bool forward)</pre>	Gibt das nächste Steuerelement innerhalb des Formulars zurück.
<pre>public virtual Size GetPreferred- Size(Size proposedSize)</pre>	Liefert die Größe eines rechteckigen Bereichs innerhalb des Formulars zurück, in dem ein Steuerelement eingefügt werden kann.
<pre>public void Hide()</pre>	Verbirgt das Formular (macht es unsichtbar).
<pre>public void Invalidate()</pre>	Erzwingt das Neuzeichnen des Formulars.
<pre>public void PerformAutoScale()</pre>	Veranlasst die Skalierung des Formulars und aller untergeordneten Steuerelemente.
<pre>public void PerformLayout()</pre>	Erzwingt die Anwendung eines bestimmten Layouts auf das Formular und alle untergeordneten Steuerelemente.
<pre>public Point PointToClient(Point p)</pre>	Liefert die Position des durch p angegebenen Bildschirmpunktes in Fensterkoordinaten zurück.
<pre>public Point PointToScreen(Point p)</pre>	Liefert die Position des durch p angegebenen Fensterpunktes in Bildschirmkoordinaten zurück.
<pre>public void Show()</pre>	Zeigt das Formular an.
<pre>public DialogResult ShowDialog()</pre>	Zeigt das Formular als modalen Dialog an.
<pre>public void SendToBack()</pre>	Sendet das Formular in den Hintergrund.

Tabelle A.14: Übersicht über die wichtigsten Methoden der Klasse Form (Forts.)

Hinweis

Drag & Drop steht für das Ziehen von Elementen auf ein Formular, wie Sie es beispielsweise innerhalb des Designers mit Elementen aus der TOOLBOX machen. Dabei ziehen Sie ein Steuerelement auf das Formular im Designer.

Wenn Sie ein Fenster mithilfe der Methode **Show** aufrufen, kann das Fenster verschoben oder minimiert werden. Sie können somit die anderen Fenster Ihrer Anwendung weiterhin bedienen. Dies hat den Nachteil, dass Sie nicht wissen, welche Taste aktuell gedrückt wurde. Weiterhin kann es vorkommen, dass dadurch das Fenster durch andere Fenster verdeckt wird. Fenster und Dialoge, die mithilfe der Methode **Show** angezeigt werden, werden als **nichtmodale Fenster** bzw. Dialoge bezeichnet.

Wenn Sie dagegen ein Fenster im Sinne eines Dialogs mithilfe der Methode **ShowDialog** aufrufen, bleibt der Fokus auf dem gezeigten Fenster. Sie können erst dann wieder mit dem Rest der Anwendung interagieren, wenn das mit **ShowDialog** aktuell gezeigte Fenster geschlossen wird. Ein solches Fenster bzw. ein solcher Dialog wird als **modales Fenster** bzw. **modaler Dialog** bezeichnet.

Das Ergebnis der Methode **ShowDialog** ist ein Wert der Enumeration **DialogResult**, die die folgenden Members definiert:

- **Abort**: Die Abbruchtaste wurde betätigt.
- **Cancel**: Die Taste ABBRECHEN wurde betätigt.
- **Ignore**: Die Taste IGNORIEREN wurde betätigt.
- **No**: Die Taste NEIN wurde betätigt. Dies ist nur im Zusammenhang mit JA/NEIN- oder JA/NEIN/ABBRECHEN-Dialogen sinnvoll.
- **None**: Es wurde keine Taste betätigt.
- **OK**: Die Taste OK wurde betätigt.
- **Retry**: Die Taste WIEDERHOLEN wurde betätigt.
- **Yes**: Die Taste JA wurde betätigt. Dies ist wie der Member **No** nur im Zusammenhang mit JA/NEIN- oder JA/NEIN/ABBRECHEN-Dialogen sinnvoll.

Um dies zu demonstrieren, erzeugen Sie ein eigenes Fenster mit den Tasten OK und ABBRECHEN. Dabei setzen Sie für die Eigenschaft **DialogResult** des Fensters bei der Taste OK den Wert **DialogResult.OK** und für die Taste ABBRECHEN den Wert **DialogResult.Cancel**:

```
public partial class LoginDialog : Form
{
    public LoginDialog()
    {
        InitializeComponent();
    }

    private void btnOk_Click(object sender, EventArgs e)
    {
        //Taste OK betätigt, Ergebnis ist OK
        DialogResult = DialogResult.OK;
    }

    private void btnCancel_Click(object sender, EventArgs e)
```

```

{
    //Taste Abbrechen betätigt, Ergebnis ist Abbrechen
    DialogResult = DialogResult.Cancel;
}
}
    
```

Listing A.4: Setzen des Ergebnisses für das Dialogfenster

Wenn Sie nun innerhalb Ihres Hauptformulars den Dialog anzeigen und erfahren wollen, welche Taste gedrückt wurde, müssen Sie das Ergebnis der Methode **ShowDialog** betrachten:

```

public partial class Form1 : Form
{
    ...
    private void btnLogin_Click(object sender, EventArgs e)
    {
        //Erzeugung des Dialogs
        LoginDialog login = new LoginDialog();

        //Prüfen der Rückgabe nach Schließen des Dialogs
        if(login.ShowDialog() == DialogResult.OK)
        {
            //Die Taste OK wurde betätigt
            lblResult.Text = "OK betätigt";
        }
        else
        {
            //Die Taste Abbrechen wurde betätigt
            lblResult.Text = "Abbrechen betätigt";
        }
    }
}
    
```

Listing A.5: Öffnen und Abfragen des Ergebnisses eines eigenständigen Dialogs

Das entsprechende Beispiel sieht folgendermaßen aus:



Abb. A.5: Beispiel eines modalen Fensters

A.1.3 Designer vs. Code

Wie Sie bereits wissen, gibt es zwei Möglichkeiten, um dem Formular Steuerelemente hinzuzufügen bzw. die Eigenschaften eines Formulars oder Steuerelements zu ändern: Entweder Sie verwenden den Designer aus Visual Studio, oder Sie nutzen die Codebehind-Datei der Formularklasse und fügen dem Formular mittels Codeanweisungen neue Steuerelemente hinzu. Sie fragen sich nun bestimmt, wann welche Methode die Sinnvollere darstellt.

Die Antwort darauf ist nicht immer einfach, Sie können sich jedoch an folgenden Tipps orientieren:

- Wollen Sie dem Formular Steuerelemente dynamisch hinzufügen bzw. wollen Sie Eigenschaften des Formulars oder eines Steuerelements dynamisch zur Laufzeit über bestimmte Bedingungen ändern, empfiehlt sich die Variante im Code.
- Wollen Sie das Layout für die Steuerelemente dynamisch festlegen, empfiehlt sich die Festlegung im Code.
- Wenn Sie Steuerelemente und Eigenschaften zu Beginn fest definieren wollen, können Sie den Designer verwenden.

Hinweis

Wir werden in diesem Kapitel überwiegend mit dem Designer arbeiten.

Zu bedenken ist, dass, wenn Sie Steuerelemente grundsätzlich über den Designer hinzufügen, deren Position auf dem Formular **absolut** ist. Wenn Sie dem Formular beispielsweise ein Steuerelement des Typs `BUTTON` hinzufügen, anschließend die Anwendung starten und das Fenster dann vergrößern oder verkleinern, werden Sie feststellen, dass sich die Schaltfläche nicht automatisch dem neuen Verhältnis anpasst:

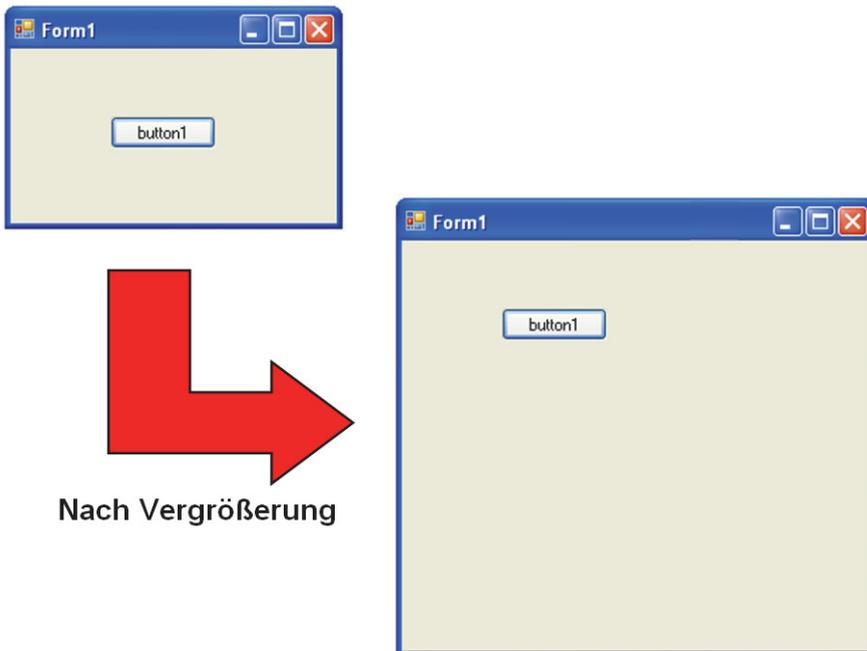


Abb. A.6: Problem der absoluten Positionierung von Steuerelementen

Dies kann entweder durch Positionierung innerhalb des Codes erfolgen oder mithilfe von Layoutelementen, die Sie in Abschnitt A.4 kennenlernen werden.

A.1.4 Für Ereignisse registrieren

Jetzt, nachdem Sie erfahren haben, wie Sie die Eigenschaften eines Formulars oder Steuerelements mithilfe des EIGENSCHAFTENFENSTERS anpassen können, ist es noch wichtig zu wissen, wie Sie sich für gewisse Ereignisse des Formulars bzw. des Steuerelements registrieren können. Ein solches Ereignis ist beispielsweise das Ereignis `Click` der Klasse `Button`, um auf einen Mausklick auf die Schaltfläche reagieren zu können. Sie können zur Registrierung entweder den Code direkt verwenden oder innerhalb des EIGENSCHAFTENFENSTERS auf den Reiter **EREIGNISSE**  klicken. Daraufhin werden Ihnen sämtliche Ereignisse angezeigt, für die Sie sich registrieren können:

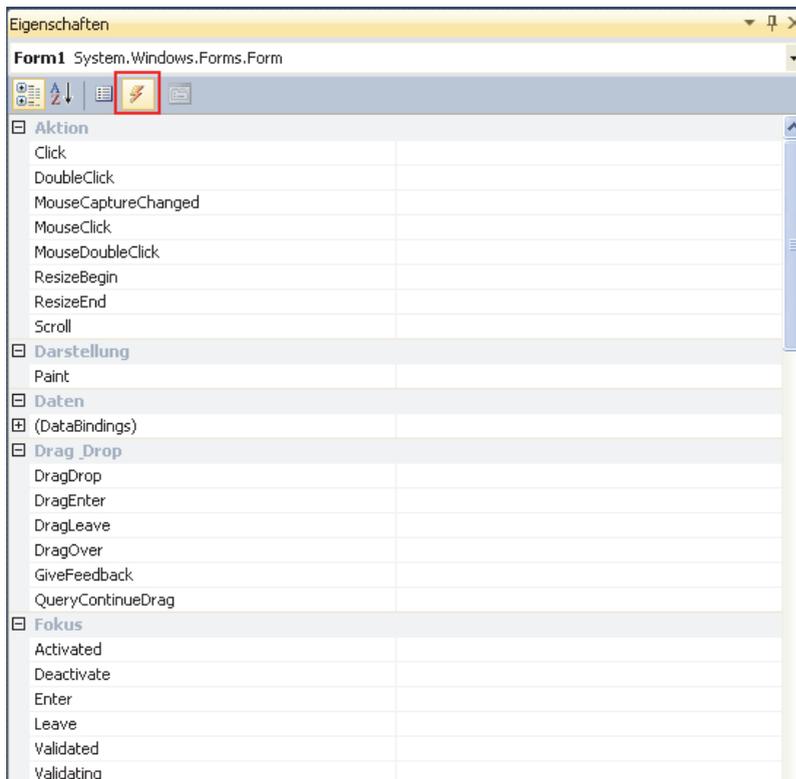


Abb. A.7: Ansicht innerhalb des EIGENSCHAFTENFENSTERS zur Registrierung für Ereignisse

Wenn Sie einen Doppelklick auf eins der Ereignisse durchführen, wird automatisch ein Eventhandler in der Codebehind-Datei des Formulars erzeugt, in dem Sie auf das entsprechende Ereignis reagieren können. Dabei wird eine Standardbezeichnung vergeben. Sie können auch einen Namen für den Eventhandler des Ereignisses eingeben und ihn mittels der Taste  in der Codebehind-Datei erzeugen. Bevor Sie sich nun den einzelnen Steuerelementen zuwenden, sollen Sie zunächst einmal die wichtigsten Ereignisse für ein Formular kennenlernen. Die folgende Tabelle zeigt die wichtigsten Ereignisse der Klasse `Form` bzw. der von `Form` abgeleiteten Klasse:

Ereignis	Bedeutung
Activated	Dieses Ereignis tritt ein, wenn das Formular aktiviert wurde.
Click	Dieses Ereignis tritt ein, wenn auf das Formular geklickt wird.
Closed	Dieses Ereignis tritt ein, wenn das Formular geschlossen wurde.
Closing	Dieses Ereignis tritt ein, wenn das Formular gerade geschlossen wird.
Deactivate	Dieses Ereignis tritt ein, wenn das Formular deaktiviert ist.
DragEnter	Dieses Ereignis tritt ein, wenn Daten auf das Formular gezogen werden.
DragOver	Dieses Ereignis tritt ein, wenn sich die zu ziehenden Daten über dem Formular befinden.
DragLeave	Dieses Ereignis tritt ein, wenn der Drag & Drop-Vorgang beendet wurde, indem die zu ziehenden Daten auf dem Formular abgelegt werden.
Enter	Dieses Ereignis tritt ein, wenn in den Bereich des Formulars eingetreten wird.
FormClosed	Dieses Ereignis tritt ein, nachdem das Formular geschlossen wurde.
FormClosing	Dieses Ereignis tritt ein, während das Formular geschlossen wird.
GotFocus	Dieses Ereignis tritt ein, wenn das Formular den Fokus erhält.
KeyDown	Dieses Ereignis tritt ein, wenn eine Taste der Tastatur gedrückt wird, während das Formular den Fokus hat.
KeyUp	Dieses Ereignis tritt ein, wenn eine Taste der Tastatur losgelassen wird, während das Formular den Fokus hat.
KeyPress	Dieses Ereignis tritt ein, wenn eine Taste der Tastatur gedrückt wird, während das Formular den Fokus hat.
Load	Dieses Ereignis tritt ein, bevor das Formular zum ersten Mal angezeigt wird.
LostFocus	Dieses Ereignis tritt ein, wenn das Formular den Fokus verliert.
Resize	Dieses Ereignis tritt ein, wenn die Größe des Formulars verändert wurde.

Tabelle A.15: Übersicht über die wichtigsten Ereignisse der Klasse Form

Ereignis	Bedeutung
Move	Dieses Ereignis tritt ein, wenn das Formular verschoben wurde.
Paint	Dieses Ereignis tritt ein, wenn das Formular neu gezeichnet wird.
Shown	Dieses Ereignis tritt ein, wenn das Formular angezeigt wird.
HelpRequested	Dieses Ereignis tritt ein, wenn der Anwender mittels der Taste F1 Hilfe anfordert.

Tabelle A.15: Übersicht über die wichtigsten Ereignisse der Klasse Form (Forts.)

Die folgende Abbildung zeigt die Reihenfolge der Ereignisse beim Laden eines Formulars:

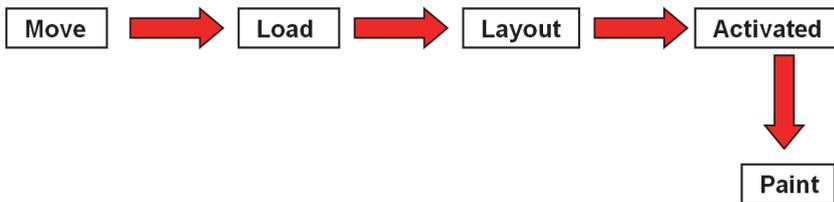


Abb. A.8: Ereignisreihenfolge beim Laden des Formulars

Beim Schließen des Formulars werden folgende Ereignisse der Reihe nach ausgelöst:



Abb. A.9: Ereignisreihenfolge beim Schließen des Formulars

A.2 Allgemeine Steuerelemente

Um das Formular um weitere Steuerelemente zu ergänzen, können Sie sie entweder per Designer auf das Formular ziehen oder Sie erzeugen sie im Code und fügen sie anschließend der Eigenschaft **Controls** der Klasse **Form** hinzu:

```

public partial Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        //Hinzufügen eines Buttons
        Button button = new Button(){Text = "Klick mich" };
        this.Controls.Add(button);
    }
}
  
```

Listing A.6: Beispielerzeugung eines Steuerelements innerhalb der Codebehind-Datei

Im Folgenden sollen die wichtigsten Steuerelemente, die häufig in der Praxis verwendet werden, genauer betrachtet werden. Dabei werden die wichtigsten Eigenschaften und Ereignisse der jeweiligen Steuerelemente definiert, die sich nur auf das entsprechende Steuerelement beziehen und keine Gemeinsamkeiten mit anderen Steuerelementen haben.

Wichtig

Da viele Anwender mit Shortcuts und der Tabulatortaste arbeiten, ist die Eigenschaft **TabIndex** besonders wichtig, weil Sie damit für jedes Steuerelement die Tabulatorreihenfolge definieren.

A.2.1 Button

Eine Schaltfläche zum Klicken wird mithilfe des Steuerelements **Button** erzeugt. Ein einfacher **Button** hat das folgende Aussehen:



Die Klasse **Button** definiert die folgenden Eigenschaften und Ereignisse:

Darstellung und Bezeichnung des Buttons festlegen

Eigenschaft	Bedeutung
FlatStyle	Legt die Darstellung der Schaltfläche fest, wenn ein Benutzer die Maus über das Steuerelement bewegt bzw. darauf klickt.
Text	Legt den Text fest, der auf der Schaltfläche angezeigt wird.
TextAlign	Legt die Ausrichtung des Textes auf der Schaltfläche fest. Die Werte entsprechen denen der Eigenschaft ImageAlign .
Margin	Legt den Abstand der Schaltfläche zum Formular für die folgenden Richtungen separat fest: links, oben, rechts, unten.
Visible	Legt fest, ob die Schaltfläche sichtbar ist.
Name	Legt den Namen der Instanz der Schaltfläche zur Verwendung im Code fest.

Tabelle A.16: Die wichtigsten Eigenschaften zur Festlegung der Darstellung eines Buttons

Bilder für Buttons festlegen

Eigenschaft	Bedeutung
ImageAlign	Legt die Ausrichtung des Bildes auf der Schaltfläche fest. Dabei sind die folgenden Werte möglich: TopLeft, TopCenter, TopRight, MiddleLeft, MiddleCenter, MiddleRight, BottomLeft, BottomCenter, BottomRight.

Tabelle A.17: Die wichtigsten Eigenschaften zur Festlegung der Bilder eines Buttons

Eigenschaft	Bedeutung
ImageIndex	Legt den Index des Bildes in Form einer Ganzzahl für die <code>ImageList</code> fest.
ImageKey	Legt den Index des Bildes in Form eines Schlüssels für die <code>ImageList</code> fest.
ImageList	Definiert die Liste von Bildern, die auf der Schaltfläche angezeigt werden sollen.
TextImageRelation	Legt die Position des Bildes in Relation zum Text auf der Schaltfläche fest. Folgende Werte sind möglich: <ul style="list-style-type: none"> ■ Overlay: Bild und Text nehmen den gleichen Raum auf der Schaltfläche ein. ■ ImageAboveText: Das Bild wird vertikal über dem Text angezeigt. ■ TextAboveImage: Der Text wird vertikal über dem Bild angezeigt. ■ ImageBeforeText: Das Bild wird horizontal vor dem Text angezeigt. ■ TextBeforeImage: Der Text wird horizontal vor dem Bild angezeigt.

Tabelle A.17: Die wichtigsten Eigenschaften zur Festlegung der Bilder eines Buttons (Forts.)

Mausereignisse für einen Button festlegen

Ereignis	Bedeutung
Click	Dieses Ereignis tritt ein, wenn der Anwender mittels Maus auf die Schaltfläche klickt.
MouseClick	Dieses Ereignis tritt ein, wenn der Anwender mittels Maus auf die Schaltfläche klickt.
MouseDown	Dieses Ereignis tritt ein, wenn der Mauszeiger über der Komponente ist und eine Maustaste gedrückt wird.
MouseEnter	Dieses Ereignis tritt ein, wenn der Mauszeiger in den Bereich der Schaltfläche eintritt.
MouseHover	Dieses Ereignis tritt ein, wenn der Mauszeiger über der Schaltfläche ist.
MouseLeave	Dieses Ereignis tritt ein, wenn der Mauszeiger aus dem Bereich der Schaltfläche austritt.
MouseMove	Dieses Ereignis tritt ein, wenn der Mauszeiger über der Schaltfläche bewegt wird.
MouseUp	Dieses Ereignis tritt ein, wenn sich der Mauszeiger auf der Schaltfläche befindet und eine Maustaste losgelassen wird.

Tabelle A.18: Die wichtigsten Ereignisse zur Reaktion auf Mausereignisse der Klasse `Button`

Welche Auswirkungen das Setzen der Eigenschaft **FlatStyle** auf das Aussehen des Steuerelements hat, zeigt die folgende Abbildung:

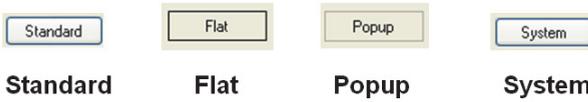


Abb. A.10: Unterschiede beim Setzen der Eigenschaft FlatStyle

Das folgende Beispiel zeigt die Verwendung des Steuerelements **Button**. Dabei soll nach einem Klick auf die Schaltfläche der Text GEKLICKT auf der Schaltfläche angezeigt werden. Wählen Sie als Instanznamen die Bezeichnung btnTest:

```
public partial class Form1 : Form
{
    ...
    //Eventhandler für das Ereignis Click des Buttons
    private btnTest_Click(object sender, EventArgs e)
    {
        btnTest.Text = "Geklickt";
    }
}
```

Listing A.7: Beispielverwendung des Steuerelements Button

A.2.2 CheckBox

Das Steuerelement **CheckBox** wird verwendet, um eine Option festzulegen. Dabei wird mittels des Häkchens angegeben, ob die Option ausgewählt ist. Das Steuerelement sieht grundlegend so aus:



Im Folgenden werden die wichtigsten Eigenschaften und Ereignisse des Steuerelements erläutert:

Darstellung der CheckBox

Eigenschaft	Bedeutung
Appearance	Legt die Darstellung des Steuerelements fest. Folgende Angaben sind möglich: <ul style="list-style-type: none"> ■ Normal: Die Standarddarstellung ■ Button: Darstellung als Button
CheckAlign	Legt die Position des Kontrollkästchens fest. Folgende Werte sind möglich: TopLeft, TopCenter, TopRight, MiddleLeft, MiddleCenter, MiddleRight, BottomLeft, BottomCenter, BottomRight.
FlatStyle	Entspricht der Eigenschaft des Steuerelements Button.

Tabelle A.19: Die wichtigsten Eigenschaften zur Festlegung der Darstellung einer CheckBox

Eigenschaft	Bedeutung
Text	Legt den anzuzeigenden Text fest.
Enabled	Legt fest, ob das Steuerelement aktiviert ist.
Name	Legt den Instanznamen innerhalb des Codes fest.
Visible	Legt fest, ob das Steuerelement sichtbar ist.

Tabelle A.19: Die wichtigsten Eigenschaften zur Festlegung der Darstellung einer CheckBox

Aktivierungsstatus der CheckBox ermitteln

Eigenschaft/Ereignis	Bedeutung
Checked	Legt fest, ob das Kontrollkästchen aktiviert wurde.
CheckState	Legt den Zustand des Steuerelements fest. Folgende Werte sind möglich: <ul style="list-style-type: none"> ■ Unchecked: Das Kontrollkästchen ist nicht aktiviert. ■ Checked: Das Kontrollkästchen ist aktiviert. ■ Indeterminate: Der Zustand des Kontrollkästchens ist unbestimmt.
ThreeState	Legt fest, ob das Steuerelement drei anstatt zwei Zustände zulässt.
Click	Dieses Ereignis tritt ein, wenn auf das Steuerelement geklickt wird.
CheckedChanged	Dieses Ereignis tritt ein, wenn das Kontrollkästchen aktiviert bzw. deaktiviert wird.
CheckStateChanged	Dieses Ereignis tritt ein, wenn der Zustand der CheckBox geändert wird.

Tabelle A.20: Die wichtigsten Eigenschaften und Ereignisse zur Ermittlung des Aktivierungsstatus einer CheckBox

Die folgende Abbildung zeigt die unterschiedlichen Darstellungen beim Setzen der Eigenschaft **Appearance**:



Abb. A.11: Unterschiede beim Festlegen der Eigenschaft Appearance

Das folgende Beispiel demonstriert die Verwendung des Steuerelements:

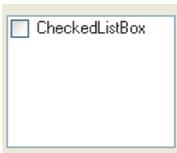
```
public partial class Form1 : Form
{
    ...
    //Eventhandler des Ereignisses CheckStateChanged
    private void chkBxTest_CheckedChanged(object sender, EventArgs e)
    {
        //Wurde das Kontrollkästchen aktiviert?
    }
}
```

```
if(chkBxTest.Checked)
    chkBxTest.Text = "Ausgewählt";
else
    chkBx.Text = "Abgewählt";
}
```

Listing A.8: Beispielverwendung des Steuerelements CheckBox

A.2.3 CheckedListBox

Um beispielsweise eine Liste von Auswahlmöglichkeiten anzubieten, empfiehlt sich das Steuerelement **CheckedListBox**. Es definiert eine Kombination aus dem Steuerelement **Listbox** zum Anzeigen einer Liste und den einzelnen **CheckBox**-Steuerelementen. Die Grunddarstellung der **CheckedListBox** sieht folgendermaßen aus:



Die folgende Tabelle zeigt die wichtigsten Eigenschaften und Ereignisse des Steuerelements:

Eigenschaft/Ereignis	Bedeutung
BorderStyle	Legt die Darstellung des Rahmens der Liste fest. Folgende Werte sind möglich: Fixed3D, None, FixedSingle.
ThreeDCheckBoxes	Legt fest, ob die Darstellung der Kontrollkästchen flach oder normal ist.
Items	Legt die Elemente des Steuerelements fest. Im EIGENSCHAFTENFENSTER kann über die angegebene Schaltfläche ein neues Fenster geöffnet werden, mittels dem zeilenweise die Einträge festgelegt werden können.
FormatString	Legt die Formatierung der anzuzeigenden Werte fest. Mittels der im EIGENSCHAFTENFENSTER hinterlegten Schaltfläche bei der Eigenschaft FormatString kann ein Fenster zur Definition der Formatierung angezeigt werden.
CheckOnClick	Legt fest, ob der Aktivierungszustand des Kontrollkästchens mit dem ersten Klick auf ein Element erfolgen soll.
ColumnWidth	Legt fest, wie breit eine Spalte in einer mehrspaltigen Listbox ist.
HorizontalScrollbar	Legt fest, ob eine horizontale Laufleiste angezeigt werden soll.
MultiColumn	Legt fest, ob die Listbox aus mehreren Spalten bestehen kann.
ScrollAlwaysVisible	Legt fest, ob die Laufleiste immer sichtbar ist.

Tabelle A.21: Übersicht der wichtigsten Eigenschaften und Methoden der Klasse **CheckedListBox**

Eigenschaft/Ereignis	Bedeutung
SelectionMode	Legt den Modus für die Auswahl der Elemente fest: <ul style="list-style-type: none"> ■ One: Nur jeweils ein Element kann ausgewählt werden. ■ None: Kein Element kann ausgewählt werden. ■ MultiSimple: Mehrere Elemente können ausgewählt werden. ■ MultiExtended: Mehrere Elemente können mithilfe der <input type="checkbox"/> <input type="checkbox"/>, <input type="checkbox"/> <input type="checkbox"/> und den Pfeiltasten ausgewählt werden.
Sorted	Legt fest, ob die Liste sortiert werden soll.
SelectedIndexChanged	Dieses Ereignis tritt ein, wenn sich die Auswahl eines Elements der Liste ändert.
ItemCheck	Dieses Ereignis tritt ein, wenn ein Element aus der Liste ausgewählt/abgewählt wird.

Tabelle A.21: Übersicht der wichtigsten Eigenschaften und Methoden der Klasse `CheckedListBox` (Forts.)

Das nun folgende Beispiel soll die Verwendung des Steuerelements verdeutlichen. Zunächst wird eine `CheckedListBox` erzeugt. Mittels des Codes werden drei Einstellungsoptionen zur Auswahl hinzugefügt. Das Hinzufügen erfolgt über die Methode `Add` zur Eigenschaft `Items`, die eine Aufzählung von Elementen darstellt. Anschließend findet die Auswertung des Ereignisses `ItemCheck` statt:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        //Hinzufügen mehrerer Auswahlmöglichkeiten zur Liste
        chkListBox.Items.Add("Erste Option");
        chkListBox.Items.Add("Zweite Option");
        chkListBox.Items.Add("Dritte Option");
    }

    //Eventhandler für das Ereignis ItemCheck
    private void chkListBox_ItemCheck(object sender,
        ItemCheckEventArgs e)
    {
        //Wurde das aktuelle Element ausgewählt?
        if(e.NewValue == CheckState.Checked)
        {
            chkListBox.Items[e.Index] =
                chkListBox.Items[e.Index] + " ausgewählt";
        }
        else
        {
            chkListBox.Items[e.Index] =
                chkListBox.Items[e.Index] + " abgewählt";
        }
    }
}
```

Listing A.9: Beispielverwendung des Steuerelements `CheckedListBox`

A.2.4 ComboBox

Das Steuerelement **ComboBox** ist in solchen Situationen sinnvoll, wenn Sie eine Auswahl anbieten möchten, bei der nur ein Element gleichzeitig ausgewählt sein kann. Die Grunddarstellung einer **ComboBox** sieht folgendermaßen aus:



Die folgende Tabelle zeigt die wichtigsten Eigenschaften und Ereignisse des Steuerelements:

Eigenschaft/Ereignis	Bedeutung
DropDownStyle	Legt die Darstellung des Steuerelements fest. Mögliche Werte sind: <ul style="list-style-type: none">■ Simple: Die Anzeige beschränkt sich auf die Bearbeitung des Textteils. Es können neue Werte eingegeben werden.■ DropDown: Es wird ein DropDown-Menü angezeigt, bei dem der Anwender neue Werte eintragen kann.■ DropDownList: Es wird ein DropDown-Menü angezeigt, bei dem der Anwender keine neuen Werte eintragen kann.
Items	Legt die Elemente der ComboBox fest.
Text	Legt den anzuzeigenden Text fest.
AutoCompleteSource	Legt die Quelle für die automatische Vervollständigung der Liste fest. Wollen Sie beispielsweise innerhalb der Listenelemente der ComboBox suchen, müssen Sie als Wert ListItems festlegen.
AutoCompleteMode	Legt das Verhalten für die automatische Vervollständigung fest. Die folgenden Angaben sind möglich: <ul style="list-style-type: none">■ None: Keine Vervollständigung■ Suggest: Schlägt eine oder mehrere Möglichkeiten vor.■ Append: Fügt die Zeichen der am ehesten infrage kommenden Zeichenfolge zu den bisherigen Zeichen im ComboBox-Feld hinzu.■ SuggestAppend: Wendet die Optionen Suggest und Append gleichzeitig an.
DropDownHeight	Legt die Höhe des DropDown -Menüs fest.
DropDownWidth	Legt die Breite des DropDown -Menüs fest.
ItemHeight	Legt die Höhe jedes Elements der ComboBox fest.
MaxDropItems	Legt fest, wie viele Elemente innerhalb der ComboBox angezeigt werden können.

Tabelle A.22: Übersicht über die wichtigsten Eigenschaften und Methoden der Klasse **ComboBox**

Eigenschaft/Ereignis	Bedeutung
DropDown	Dieses Ereignis tritt ein, wenn das DropDown -Menü angezeigt wird.
DropDownClosed	Dieses Ereignis tritt ein, wenn das DropDown -Menü geschlossen wurde.
SelectedIndexChanged	Dieses Ereignis tritt ein, wenn ein Element der ComboBox ausgewählt wurde.

Tabelle A.22: Übersicht über die wichtigsten Eigenschaften und Methoden der Klasse `ComboBox` (Forts.)

Das nun folgende Beispiel soll die Verwendung des Steuerelements verdeutlichen. Dabei wird dem Formular ein Steuerelement des Typs `ComboBox` mit der Eigenschaft `NAME cmbxOptions` und den Eigenschaften `AUTOCOMPLETEMODE Append` und `AUTOCOMPLETESOURCE ListItems` hinzugefügt. Weiterhin wird eine Registrierung für das Ereignis `SelectedIndexChanged` durchgeführt. Zum Anzeigen des ausgewählten Eintrags der `ComboBox` wird ein `LABEL` mit der Eigenschaft `NAME lblResult` verwendet. Die Einträge werden im Konstruktor des Formulars der `ComboBox` hinzugefügt:

```
public partial class Form1 : Form
{
    ...
    public Form1()
    {
        InitializeComponent();
        //Hinzufügen der Einträge
        cmbxOptions.Items.Add("Erste Option");
        cmbxOptions.Items.Add("Zweite Option");
        cmbxOptions.Items.Add("Dritte Option");
    }

    //Eventhandler des Ereignisses SelectedIndexChanged
    private void SelectedIndexChanged(object sender,
        EventArgs e)
    {
        //Wurde etwas ausgewählt?
        if(cmbxOptions.SelectedIndex >= 0)
        {
            lblResult.Text = (string)cmbxOptions.SelectedItem;
        }
    }
}
```

Listing A.10: Beispielverwendung des Steuerelements `ComboBox`

A.2.5 DateTimePicker

Wollen Sie dem Anwender eine Auswahl für das Datum anbieten, dann sollten Sie das Steuerelement `DateTimePicker` verwenden:



Abb. A.12: Das Steuerelement DateTimePicker

Dieses Steuerelement besteht aus einer ComboBox, die das aktuelle Datum anzeigt. Wenn Sie die ComboBox aufklappen, wird ein Kalender angezeigt, wie Sie ihn von Windows kennen, in dem Sie ein anderes Datum auswählen können. Das Steuerelement besitzt die folgenden Eigenschaften und Ereignisse:

Eigenschaft/Ereignis	Bedeutung
Format	Legt das Format fest, in dem das Datum und die aktuelle Uhrzeit angezeigt werden sollen. Die folgenden Werte sind möglich: <ul style="list-style-type: none"> ■ Long: Zeigt den Tag und das Datum an. ■ Short: Zeigt nur das Datum an. ■ Time: Zeigt nur die aktuelle Zeit an. ■ Custom: Legt ein benutzerdefiniertes Format fest.
MaxDate	Legt das maximale Datum für die Auswahl fest.
MinDate	Legt das minimale Datum für die Auswahl fest.
ShowCheckBox	Legt fest, ob eine zusätzliche CheckBox zur Auswahl angezeigt werden soll.
ShowUpDown	Legt fest, ob die Auswahl anhand eines Dropdown-Menüs oder anhand eines NumericUpDown erfolgt.
Value	Legt das aktuell anzuzeigende Datum bzw. die aktuell anzuzeigende Zeit fest.
ValueChanged	Dieses Ereignis tritt ein, wenn der aktuelle Wert geändert wurde.

Tabelle A.23: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements DateTimePicker

Für das aktuelle Steuerelement bedarf es keines Beispiels, da es selbsterklärend ist.

A.2.6 Label

Wenn Sie hilfreiche Beschriftungen auf Ihrem Formular platzieren wollen, müssen Sie das Steuerelement **Label** verwenden:



Abb. A.13: Ein einfaches Label

Das Steuerelement hat die folgenden Eigenschaften und Ereignisse:

Eigenschaft/Ereignis	Bedeutung
BackColor	Legt die Hintergrundfarbe fest.
BorderStyle	Entspricht dem Verhalten des Steuerelements <code>CheckedListBox</code> .
FlatStyle	Entspricht dem Verhalten des Steuerelements <code>CheckBox</code> .
Image	Legt ein Hintergrundbild für das Steuerelement fest.
Text	Legt den anzuzeigenden Text fest.
TextAlign	Legt die Ausrichtung des Textes fest. Mögliche Werte sind: TopLeft, TopCenter, TopRight, MiddleLeft, MiddleCenter, MiddleRight, BottomLeft, BottomCenter, BottomRight .
Click	Dieses Ereignis tritt ein, wenn auf das Label geklickt wurde.
TextChanged	Dieses Ereignis tritt ein, wenn sich der Text geändert hat.

Tabelle A.24: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements `Label`

Auch in diesem Fall bedarf es keines Beispiels, da das Steuerelement selbsterklärend ist.

A.2.7 LinkLabel

Das Steuerelement `LinkLabel` ist im Prinzip ein einfaches `Label`. Es erweitert jedoch das Steuerelement um die Möglichkeit, dass bei einem Klick auf das Steuerelement eine Weiterleitung zu einem hinterlegten Link erfolgt, wie Sie es aus dem Internet gewohnt sind.

[LinkLabel](#)

Abb. A.14: Das Steuerelement `LinkLabel`

Die folgende Tabelle zeigt die wichtigsten Eigenschaften und Ereignisse für das Steuerelement:

Eigenschaft/Ereignis	Bedeutung
ActiveLinkColor	Legt die Farbe für den Link fest, wenn der Anwender auf diesen klickt.
BackColor	Legt die Hintergrundfarbe fest.
DisabledLinkColor	Legt die Farbe für den Link fest, wenn dieser deaktiviert ist.
LinkArea	Legt den Teil des Textes der Eigenschaft <code>Text</code> fest, der als Link dargestellt werden soll. Dabei werden der Startindex und die Länge des Strings angegeben.

Tabelle A.25: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements `LinkLabel`

Eigenschaft/Ereignis	Bedeutung
LinkBehavior	Legt das Verhalten des Links fest. Mögliche Werte sind: <ul style="list-style-type: none">■ SystemDefault: Das Standardverhalten■ AlwaysUnderline: Der Link ist immer unterstrichen.■ HoverUnderline: Der Link wird unterstrichen, sobald der Benutzer mit der Maus über dem Link steht.■ NeverUnderline: Der Link wird niemals unterstrichen.
LinkColor	Legt die Standardfarbe des Links fest.
LinkVisited	Legt fest, ob der Link als besucht markiert sein soll.
Text	Legt den Text des Labels fest.
VisitedLinkColor	Legt die Farbe des Links dem Anklicken fest.
LinkClicked	Dieses Ereignis tritt ein, wenn der Link angeklickt wurde.

Tabelle A.25: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements `LinkLabel`

Leider existiert keine Eigenschaft `Link`, mit der Sie den Link zu einer Seite festlegen können. Dies erfolgt über die Eigenschaft `Links`. Das folgende Beispiel definiert einen Link auf die Seite von Google. Sobald der Link angeklickt wird, wird die Seite aufgerufen. Zuvor wird dem Steuerelement über die Eigenschaft `Text` die Bezeichnung `Google` zugewiesen:

```
...
public Form1()
{
    InitializeComponent();
    //Neuen Link definieren. Dabei wird über die Zahlen festgelegt,
    //welcher Teil des Textes des Labels unterstrichen ist
    linkLabel1.Links.Add(0, linkLabel1.Text.Length, "www.google.de");
}

//Behandelt den Klick auf den Link und öffnet den Explorer mit der
//Seite
private void LinkLabel1_Clicked(object sender,
    LinkLabelLinkClickedEventArgs e)
{
    //Link wurde angeklickt
    linkLabel1.Links[linkLabel1.Links.IndexOf(e.Link)].Visited = true;
    //Weiterleiten auf die Seite Google
    System.Diagnostics.Process.Start(e.Link.LinkData.ToString());
}
}
```

Listing A.11: Definition eines Links zur Seite Google

A.2.8 ListBox

Mithilfe einer `ListBox` können Sie Elemente in einer Liste darstellen. Dabei kann der Benutzer eine oder sogar mehrere auswählen:

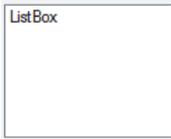


Abb. A.15: Das Steuerelement `ListBox`

Die folgende Tabelle zeigt die wichtigsten Eigenschaften und Ereignisse der `ListBox`:

Eigenschaft/Ereignis	Bedeutung
ColumnWidth	Legt die Breite in einer mehrspaltigen <code>ListBox</code> fest.
HorizontalScrollbar	Legt fest, ob der horizontale Schieberegler sichtbar ist.
Items	Legt die Elemente der <code>ListBox</code> fest. Dies ist eine Auflistung.
MultiColumn	Legt fest, ob mehrere Spalten angezeigt werden sollen.
ScrollAlwaysVisible	Legt fest, ob die horizontalen und vertikalen Schieberegler immer sichtbar sind.
SelectionMode	Legt den Auswahlmodus fest. Siehe dazu auch <code>CheckedListBox</code> .
Sorted	Legt fest, ob die Elemente sortiert angezeigt werden sollen.
SelectedIndexChanged	Dieses Ereignis tritt ein, wenn die Eigenschaft <code>SelectedIndex</code> geändert wurde. Sobald der Anwender eine Auswahl trifft, wird dieses Ereignis ausgelöst.
SelectedValueChanged	Dieses Ereignis tritt ein, wenn die Eigenschaft <code>SelectedValue</code> geändert wurde.

Tabelle A.26: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements `ListBox`

Im folgenden Beispiel wird eine `ListBox` mit Namen von Personen gefüllt. Der Benutzer kann dann einen Namen auswählen. Anschließend wird eine `MessageBox` mit dem ausgewählten Namen angezeigt:

```
...
public Form1()
{
    InitializeComponent();
    var names = new string[]{"Tim", "Tim", "Guido", "Klaus"};

    //Initialisieren der ListBox mit den Namen. Über Items.Add und
    //Items.Remove können Einträge hinzugefügt und gelöscht werden
    foreach(var name in names)
        listBox1.Items.Add(name);
}

private void ListBox_SelectedIndexChanged(object sender, EventArgs e)
{
    //Das ausgewählte Element kann über die Eigenschaft SelectedItem
    //bestimmt werden. Da der Rückgabewert vom Typ object ist, muss
```

```
//er in den zu erwarteten Typ gecastet werden
string selectedName = (string)listBox1.SelectedItem;
MessageBox.Show("Gewählt: " + selectedName);
}
```

Listing A.12: Verwendung einer ListBox zur Auswahl eines Namens

A.2.9 ListView

Die **ListView** ähnelt einer **ListBox**, mit dem Unterschied, dass den Einträgen der Liste zusätzlich Icons und **CheckBoxen** hinzugefügt werden können. Die einzelnen Einträge können Sie über die Eigenschaft **Items** ergänzen. Dies ist entweder im Programmcode oder über einen Editor möglich, den Sie durch einen Klick auf den **Button** neben der Eigenschaft **ITEMS** im **EIGENSCHAFTENFENSTER** öffnen können.

Jeder Eintrag ist vom Typ **ListViewItem** und kann wiederum eine Reihe von Einträgen enthalten. Dies können Sie festlegen, indem Sie über die Eigenschaft **SubItems** weitere Einträge hinzufügen. Jeder Eintrag besitzt die Eigenschaft **Text**, über die Sie den anzuzeigenden Text festlegen können. Für jeden Eintrag können Sie sowohl kleine als auch große Icons oder auch Icons in einer Liste verwenden. Die Icons werden in Form einer **ImageList** definiert. Diese Liste können Sie mittels eines eigenen Editors bearbeiten, wenn Sie im **EIGENSCHAFTENFENSTER** unter **IMAGELIST** auf den **Button** daneben klicken. Von der Auswahl der Einträge entspricht das Steuerelement der **ListBox** und lässt sowohl einfache als auch mehrfache Auswahl zu. Zunächst sollen nachfolgend die wichtigsten Eigenschaften, Methoden und Ereignisse des Steuerelements **ListView** betrachtet werden:

Eigenschaft/Ereignis	Bedeutung
Groups	Ermöglicht die Angabe von Gruppen zur Gruppierung von Einträgen.
View	Legt den Anzeigemodus fest. Mögliche Werte sind List , SmallIcon , LargeIcon und Details .
CheckBoxes	Legt fest, ob Checkboxes angezeigt werden sollen.
CheckItems	Liefert die aktivierten Einträge zurück.
Columns	Wird für die Detailansicht benötigt. Hierüber können Sie weitere Spalten hinzufügen.
ImageIndex	Legt den Index für das anzuzeigende Bild fest.
LabelEdit	Bestimmt, ob der Anwender den Text eines Eintrags ändern kann.
GridLines	Legt fest, ob Rasterlinien angezeigt werden sollen.
AllowColumnReorder	Legt fest, ob der Anwender die Spalten in der Detailansicht neu ordnen darf.
HeaderStyle	Legt den Stil des Headers für jede Spalte in der Detailansicht fest.
ShowGroups	Legt fest, ob die Gruppen angezeigt werden sollen.
FullRowSelect	Legt fest, ob die ganze Zeile selektiert werden kann.
ItemActivate	Dieses Ereignis tritt ein, wenn ein Eintrag ausgewählt wurde.

Tabelle A.27: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements ListView

Eigenschaft/Ereignis	Bedeutung
ColumnClick	Dieses Ereignis tritt ein, wenn auf einen Spaltenheader geklickt wurde.
ItemCheck	Dieses Ereignis tritt ein, wenn ein Eintrag über eine CheckBox verfügt und diese angeklickt wurde.

Tabelle A.27: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements `Listview`

Die wichtigsten Methoden sind **BeginUpdate** und **EndUpdate** zur Steigerung der Performance beim Aktualisieren der Liste sowie die Methode **Clone** zum Erstellen einer Kopie der Einträge. Weiterhin existieren die Methoden **GetItemAt** zur Ermittlung eines Eintrags über den Index und **EnsureVisible** zur Anzeige des gewünschten Eintrags beim Aktualisieren der Liste.

Die folgende Abbildung zeigt eine **Listview** als Detailansicht:

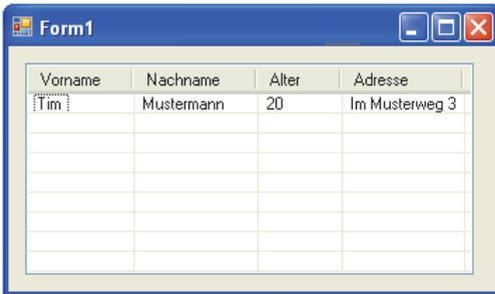


Abb. A.16: Die `Listview` als Detailansicht

A.2.10 MaskedTextBox

Um Eingaben in eine `TextBox` zu filtern, können Sie dieses spezifischere Steuerelement verwenden. Es entspricht im Wesentlichen einer `TextBox`, nur mit dem Unterschied, dass Sie zusätzlich über die Eigenschaft **Mask** eine Maske definieren.



Abb. A.17: Eine `MaskedTextBox`

Für die Definition der Maske können Sie Sonderzeichen verwenden, von denen einige optional sind. Die folgende Tabelle zeigt die Sonderzeichen:

Sonderzeichen	Bedeutung
.	Dezimaltrennzeichen für Gleitkommawerte
:	Trennzeichen für Zeitangaben
,	Trennzeichen für Tausenderstellen

Tabelle A.28: Die Sonderzeichen für die Maskierung von Eingaben

Sonderzeichen	Bedeutung
<	Alle nachfolgenden Zeichen werden in Kleinbuchstaben umgewandelt.
>	Alle nachfolgenden Zeichen werden in Großbuchstaben umgewandelt.
/	Trennzeichen für Datumsangaben
\$	Währung
&	Alphanumerisch
L	Buchstabe (klein/groß)
?	Buchstabe ist optional.
0	Zahl zwischen 0 und 9
9	Zahl oder Leerzeichen (optionale Angabe)
#	Zahl oder Leerzeichen mit Minus oder Plus (optionale Angabe)

Tabelle A.28: Die Sonderzeichen für die Maskierung von Eingaben (Forts.)

Die weiteren wichtigen Eigenschaften des Steuerelements werden in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
PromptChar	Definiert das als Platzhalter zu verwendende Zeichen.
AllowPromptAsInput	Definiert, ob das Eingabeaufforderungszeichen eine gültige Eingabe ist.
AsciiOnly	Legt fest, dass nur ASCII-Zeichen zulässig sind.
BeepOnError	Legt fest, dass bei falscher Eingabe ein Signalton ausgegeben wird.
RejectInputOnFirstFailure	Lehnt die Eingabe ab, wenn das Zeichen nicht der Maske entspricht.
MaskCompleted	Prüft, ob die Eingabe vollständig der Maske entspricht.

Tabelle A.29: Die wichtigsten Eigenschaften des Steuerelements MaskedTextBox

Im folgenden Beispiel wird eine **MaskedTextBox** definiert, die nur eine vierstellige Zahl als Eingabe akzeptiert:

```
...  
public Form1()  
{  
    InitializeComponent();  
    maskedTextBox1.PromptChar = ' ';  
    //Nur Ziffern sind möglich. Es muss eine vierstellige Zahl sein  
    maskedTextBox1.Mask = "0000";  
}
```

```
private void MaskedTextBox_TextChanged(object sender, EventArgs e)
{
    //Prüfen auf Vollständigkeit der Maske
    if(maskedTextBox1.MaskCompleted)
        MessageBox.Show("Eingabe vollständig");
}
```

Listing A.13: Definition einer MaskedTextBox, die nur Zahlen als Eingabe akzeptiert

Hinweis

Die **MaskedTextBox** kann auch für Passwordeingaben verwendet werden. Hierzu können Sie der Eigenschaft **PasswordChar** ein Zeichen angeben, das bei der Eingabe angezeigt wird.

A.2.11 MonthCalendar

Das Steuerelement **MonthCalendar** ähnelt dem Steuerelement **DateTimePicker**. Es unterscheidet sich lediglich durch die Tatsache, dass es im Gegensatz zum **DateTimePicker** nicht wie eine **ComboBox** aufgeklappt werden kann und dadurch nur den vollen Monatskalender anzeigt:



Abb. A.18: Der MonthCalendar

A.2.12 NumericUpDown

Das Steuerelement **NumericUpDown** ermöglicht das Inkrementieren und Dekrementieren eines bestehenden Wertes über zwei Scroll-Schaltflächen.



Abb. A.19: Das Steuerelement NumericUpDown

Über die Eigenschaften **Maximum** und **Minimum** können Sie den minimalen sowie den maximalen Wert festlegen, der akzeptiert wird. Die Schrittweite lässt sich über die Eigenschaft **Increment** einstellen. Wollen Sie den aktuellen Wert setzen oder abfragen, können Sie hierzu die Eigenschaft **Value** verwenden. Zu guter Letzt ist es weiterhin auch möglich, über die Eigenschaft **ThousandsSeparator** Trennzeichen für Tausenderstellen anzuzeigen. Wollen Sie auf das Klickereignis der Scrollflächen reagieren, können Sie sich für das Ereignis

Scroll registrieren. Wollen Sie dagegen lediglich eine Wertänderung ermitteln, können Sie das Ereignis **ValueChanged** verwenden.

Das folgende Beispiel prüft das Steuerelement auf eine Wertänderung. Wenn der Wert 100 erreicht ist, soll eine Nachricht ausgegeben werden:

```
...  
public Form1()  
{  
    InitializeComponent();  
    numericUpDown1.Value = 50;  
    numericUpDown1.Minimum = 0;  
    numericUpDown1.Maximum = 120;  
}  
  
private void NumericUpDown_ValueChanged(object sender, EventArgs e)  
{  
    if(numericUpDown1.Value == 100)  
        MessageBox.Show("Wert 100 erreicht");  
}
```

Listing A.14: Verwendung einer `NumericUpDown` zur Prüfung einer Wertänderung

Hinweis

Ein ähnliches Steuerelement ist das Steuerelement **DomainUpDown**. Der Unterschied ist, dass dieses Steuerelement im Gegensatz zum Steuerelement `NumericUpDown` nur Strings akzeptiert.

A.2.13 PictureBox

Bilder können Sie in einer Anwendung mittels des Steuerelements **PictureBox** anzeigen. Zur Zuweisung eines Bildes verwenden Sie die Eigenschaft **Image**. Diese erwartet eine Instanz der Klasse **Image**, von der jedoch nicht ohne Weiteres eine Instanz erzeugt werden kann. Abhilfe schafft hier die Klasse **Bitmap**, welche eine Unterklasse der Klasse **Image** ist und somit der Eigenschaft **Image** zugewiesen werden kann. Alternativ können Sie über die statische Methode **FromFile** der Klasse **Image** eine Instanz der Klasse **Image** erzeugen und diese dann der Eigenschaft der **PictureBox** zuweisen. Eine weitere Eigenschaft der **PictureBox** ist die Eigenschaft **SizeMode**, mit der Sie die Größenanpassung des Bildes steuern können. Die folgenden Werte sind möglich:

- **Normal**: Das Bild wird in der linken oberen Ecke platziert und abgeschnitten, falls es zu groß für das Steuerelement ist. Dies ist der Standardwert.
- **StretchImage**: Das Bild wird auf die Größe der **PictureBox** gestreckt bzw. verkleinert.
- **AutoSize**: Die **PictureBox** wird auf die Größe des Bildes eingestellt.
- **CenterImage**: Das Bild wird in der Mitte angezeigt, wenn die **PictureBox** größer ist. Ansonsten wird es zentriert und die Ränder werden abgeschnitten.
- **Zoom**: Das Bild wird vergrößert bzw. verkleinert. Dabei bleibt das Größenverhältnis erhalten.

Die folgende Abbildung zeigt eine **PictureBox** mit entsprechendem Bild und zugehörigem **SizeMode**:



Normal



StretchImage



AutoSize



CenterImage



Zoom

Abb. A.20: Die verschiedenen Modi der PictureBox

A.2.14 ProgressBar

Zur Anzeige des Verarbeitungsfortschritts können Sie das Steuerelement **ProgressBar** verwenden.

Die wichtigsten Eigenschaften des Steuerelements sind in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
MarqueeAnimationSpeed	Definiert die Geschwindigkeit der Animation in Millisekunden.
Minimum	Legt den minimalen Wert fest.
Maximum	Legt den maximalen Wert fest.
Step	Legt die Schrittweite fest.
Style	Legt die Darstellung fest. Folgende Werte sind möglich: <ul style="list-style-type: none"> ■ Blocks: Zeigt den Fortschritt in Blöcken an. ■ Continuous: Zeigt den Fortschritt als fortlaufende Leiste an. ■ Marquee: Zeigt den Fortschritt als andauernde Animation an.
Value	Legt den aktuellen Wert fest bzw. ruft diesen ab.

Tabelle A.30: Die wichtigsten Eigenschaften der ProgressBar

Die folgende Abbildung zeigt die unterschiedlichen Darstellungen entsprechend der **Style**-Definition:



Abb. A.21: Die verschiedenen Styles der ProgressBar

Hinweis

Unter Windows XP ist für den Style `Continuous` kein Unterschied zu `Blocks` zu erkennen. Ab Windows Vista bzw. Windows 7 sehen Sie hier einen glatten Fortschrittsbalken.

Hinweis

Sie können den aktuellen Fortschritt über die Eigenschaft `Value` festlegen. Eine andere Möglichkeit besteht in der Verwendung der Methode `PerformStep`, die bei jedem Aufruf den aktuellen Wert um den in `Step` angegebenen Wert erhöht.

A.2.15 RadioButton

Das Steuerelement `RadioButton` ähnelt dem Steuerelement `CheckBox`, jedoch kann im Gegensatz zu einer `CheckBox` nur ein `RadioButton` innerhalb einer Gruppe ausgewählt werden.

Die wichtigste Eigenschaft ist auch hier die Eigenschaft `Checked`, mit der Sie prüfen können, ob ein `RadioButton` aktiviert wurde. Weitere Eigenschaften sind `Appearance` und `FlatStyle` zur Festlegung des Aussehens sowie `Text` zur Anzeige des assoziierten Textes. Um zu prüfen, ob ein `RadioButton` ausgewählt wurde, können Sie sich für das Ereignis `CheckedChanged` registrieren. Die folgende Abbildung zeigt zwei `RadioButtons` mit unterschiedlicher Darstellung:



Abb. A.22: Zwei `RadioButtons` mit unterschiedlicher Darstellung

A.2.16 TextBox

Zur Eingabe von Werten dient das Steuerelement `TextBox`.

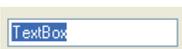


Abb. A.23: Das Steuerelement `TextBox`

Die wichtigste Eigenschaft ist **Text**, mit der Sie den anzuzeigenden Text festlegen bzw. den eingegebenen Text ermitteln können. Standardmäßig ist die **TextBox** einzelzeilig. Wollen Sie jedoch eine mehrzeilige **TextBox** definieren, müssen Sie die Eigenschaft **MultiLine** auf **true** setzen. Sollten dabei zusätzliche Bildlaufleisten benötigt werden, können Sie diese über die Eigenschaft **Scrollbars** anzeigen. Bei einer mehrzeiligen **TextBox** können Sie jede Zeile über die Eigenschaft **Lines** auslesen. Weitere wichtige Eigenschaften sind:

- **TextAlign**: Legt die Ausrichtung des Textes fest. Gültig sind die Werte **Left**, **Center**, **Right**.
- **MaxLength**: Legt die maximale Länge des Textes fest.
- **ReadOnly**: Definiert, dass keine Texteingabe möglich ist.
- **WordWrap**: Hiermit legen Sie fest, ob der Text am Ende einer Zeile umbrochen werden soll.
- **AcceptsReturn**: Ist diese Eigenschaft auf **true** gesetzt, können Sie mit der Taste  einen Zeilenumbruch einfügen.
- **PasswordChar**: Legt das Maskierungszeichen für den Fall der Verwendung als Passworteingabefeld fest.

Wollen Sie bestimmte Stellen des Textes markieren, können Sie hierfür folgende Eigenschaften verwenden:

- **SelectionStart**: Legt die Startposition der Selektion fest.
- **SelectedText**: Legt den selektierten Text fest.
- **SelectionLength**: Legt die Anzahl der markierten Zeichen fest.

Als Alternative können Sie den Text auch über die Methoden **SelectAll** für den vollständigen Text und **Select** unter Angabe der Startposition und Länge markieren.

Das Aussehen der **TextBox** lässt sich über die Eigenschaft **BorderStyle** festlegen. Dies entspricht dem Verhalten des Steuerelements **Label**. Um auf Eingabeänderungen reagieren zu können, können Sie sich für das Ereignis **TextChanged** registrieren.

A.2.17 RichTextBox

Eine umfangreichere Textverarbeitung im Sinne eines Texteditors können Sie mithilfe des Steuerelements **RichTextBox** umsetzen. Im Grunde definiert dieses Steuerelement dieselben Eigenschaften wie das Steuerelement **TextBox**. Sie können jedoch zusätzlich markierte Textabschnitte formatieren, indem Sie beispielsweise eine andere Farbe oder eine andere Schrift wählen. Hierzu existieren verschiedene **SelectionXXX**-Eigenschaften:

- **SelectionFont**: Legt die Schrift des markierten Textes fest.
- **SelectionColor**: Legt die Farbe des markierten Textes fest.
- **SelectionAlignment**: Legt die Ausrichtung des markierten Textes fest.
- **SelectionBullet**: Legt fest, ob ein Aufzählungszeichen vor den markierten Text gesetzt werden soll.

Darüber hinaus ist es sogar möglich, Dateien im *Rich Text*-Format direkt über die Methode **LoadFile** in die **RichTextBox** zu laden. Wenn Sie erfahren wollen, wann eine Auswahl stattfindet, müssen Sie sich für das Ereignis **SelectionChanged** registrieren.

Das folgende Beispiel zeigt die Verwendung der **RichTextBox**. Dabei wird ein markierter Text fett hervorgehoben und dessen Farbe auf Blau festgelegt:

```
...  
private void RichTextBox_SelectionChanged(object sender, EventArgs e)  
{  
    richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont.  
        FontFamily, 16.0F);  
    richTextBox1.SelectionColor = Color.Blue;  
}
```

Listing A.15: Formatierung eines selektierten Textes

Das Ergebnis sieht folgendermaßen aus:

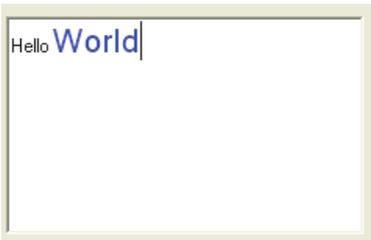


Abb. A.24: Das Ergebnis der Selektion in einer RichTextBox

A.2.18 TreeView

Wenn Sie sich den PROJEKTMAPPEN-EXPLORER in Visual Studio ansehen, erkennen Sie, dass die Dateien des Projekts in einer Baumstruktur dargestellt werden. Diese Baumstruktur können Sie über das Steuerelement **TreeView** erzeugen.

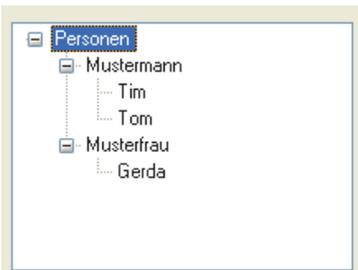


Abb. A.25: Das Steuerelement TreeView

Dabei ist es wichtig zu wissen, dass jedes Element der **TreeView** eine Instanz der Klasse **TreeNode** darstellt. Jeder Knoten besitzt eine Reihe von Unterknoten, die Sie über die Eigenschaft **Nodes** festlegen können. Dies können Sie sowohl im Programmcode als auch mittels des Editors durchführen, der sich öffnet, sobald Sie im EIGENSCHAFTENFENSTER auf die Eigenschaft **Nodes** und den erscheinenden Button klicken:

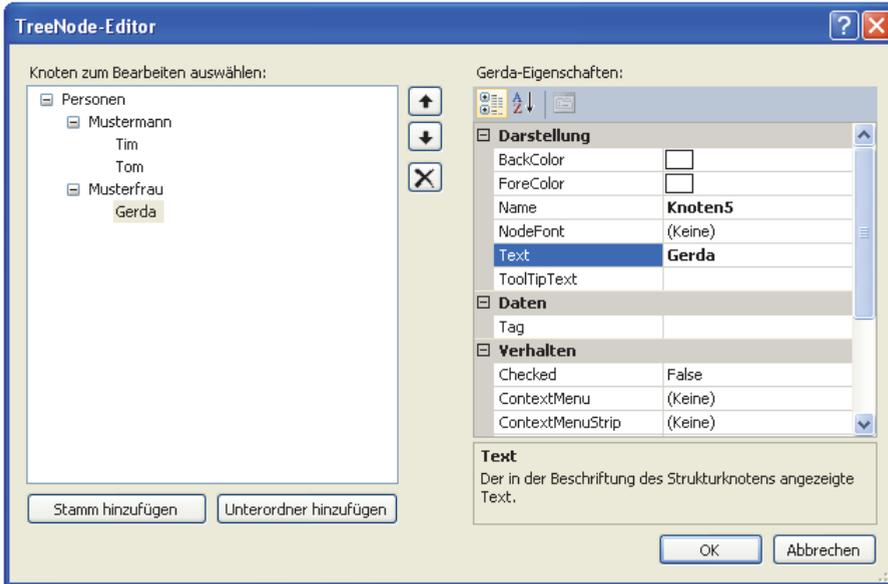


Abb. A.26: Der TreeNode-Editor

Wollen Sie denselben Baum im Programmcode zur Laufzeit erzeugen, müssen Sie die folgenden Anweisungen verwenden:

```

...
public Form1()
{
    InitializeComponent();
    //Steigerung der Performance durch Verwendung der Methoden
    //BeginUpdate und EndUpdate
    treeView1.BeginUpdate();
    treeView1.Nodes.Clear();

    //Hinzufügen des Wurzelknotens Personen
    treeView1.Nodes.Add("Personen");
    //Hinzufügen der ersten Unterebene Mustermann und Musterfrau
    treeView1.Nodes[0].Nodes.Add("Mustermann");
    treeView1.Nodes[0].Nodes.Add("Musterfrau");
    //Hinzufügen der zweiten Unterebene von Mustermann und Musterfrau
    treeView1.Nodes[0].Nodes[0].Nodes.Add("Tim");
    treeView1.Nodes[0].Nodes[0].Nodes.Add("Tom");
    treeView1.Nodes[0].Nodes[1].Nodes.Add("Gerda");
    //Abschließen der Aktualisierung
    treeView1.EndUpdate();
}

```

Listing A.16: Erzeugung eines Baums im Programmcode

Den aktuell ausgewählten Knoten erhalten Sie über das Ereignis **AfterSelect**. Der Ereignisparameter **TreeViewEventArgs** enthält die Eigenschaft **Node**, mit der Sie den aktuell ausgewählten Knoten ermitteln können.

Die wichtigsten Eigenschaften und Ereignisse von **TreeView** sind in der folgenden Tabelle dargestellt:

Eigenschaft/Ereignis	Bedeutung
CheckBoxes	Legt fest, ob CheckBoxen angezeigt werden sollen.
FullRowSelect	Bestimmt, ob die gesamte Breite farblich hervorgehoben werden soll.
HideSelection	Entfernt die Hervorhebung des aktuellen Knotens.
ImageList	Legt eine Liste für die Bilder der Knoten fest. Dabei können sowohl kleine Icons als auch große Icons festgelegt werden.
Indent	Legt die Einzugsbreite in Pixel fest.
LineColor	Legt die Farbe der Verbindungslinien fest.
LabelEdit	Legt fest, ob der Anwender den Text des Knotens ändern kann.
PathSeparator	Legt das Zeichen fest, mit dem der Pfad über die Eigenschaft FullPath getrennt wird.
ShowLines	Legt fest, ob Verbindungslinien angezeigt werden sollen.
SelectedNode	Gibt den selektierten Knoten zurück.
ShowPlusMinus	Legt fest, ob die Expander-Symbole (+, -) angezeigt werden sollen.
ShowRootLines	Legt fest, ob die Verbindungslinie zum Wurzelknoten angezeigt werden soll.
SelectedImageIndex	Legt den Index des Bildes aus der ImageList für den selektierten Knoten fest.
ImageIndex	Legt den Index des Bildes aus der ImageList für jeden nicht selektierten Knoten fest.
BeforeExpand	Dieses Ereignis tritt ein, bevor ein Knoten geöffnet wird.
AfterExpand	Dieses Ereignis tritt ein, nachdem ein Knoten geöffnet wurde.
BeforeSelect	Dieses Ereignis tritt ein, bevor ein Knoten ausgewählt wurde.
AfterSelect	Dieses Ereignis tritt ein, nachdem ein Knoten ausgewählt wurde.
BeforeCollapse	Dieses Ereignis tritt ein, bevor ein Knoten geschlossen wurde.
AfterCollapse	Dieses Ereignis tritt ein, nachdem ein Knoten geschlossen wurde.
NodeMouseClick	Dieses Ereignis tritt ein, wenn auf einen Knoten geklickt wurde.

Tabelle A.31: Die wichtigsten Eigenschaften und Ereignisse des Steuerelements **TreeView**

Weitere wichtige Methoden sind **CollapseAll** zum Schließen aller Knoten, **ExpandAll** zum Öffnen aller Knoten und **Sort** zum Sortieren aller Knoten.

Die Klasse **TreeNode**

Da alle Knoten vom Typ **TreeNode** sind, sollen auch hier kurz die wichtigsten Eigenschaften und Methoden der Klasse **TreeNode** beschrieben werden:

Eigenschaft	Bedeutung
FirstNode	Gibt den ersten Unterknoten zurück.
Index	Liefert den Index der Eigenschaft Nodes des übergeordneten Knotens zurück.
IsExpanded	Gibt einen Wahrheitswert zurück, der angibt, ob der Knoten geöffnet ist.
IsSelected	Gibt einen Wahrheitswert zurück, der angibt, ob der Knoten selektiert ist.
LastNode	Liefert den letzten Unterknoten zurück.
NextNode	Liefert den nächsten gleichrangigen Knoten zurück.
Parent	Liefert den übergeordneten Knoten zurück.
PrevNode	Liefert den vorhergehenden und gleichrangigen Knoten zurück.

Tabelle A.32: Die wichtigsten Eigenschaften der Klasse `TreeNode`

Zu den wichtigsten Methoden zählen die Methoden **Collapse** zum Schließen des Knotens, **Expand** zum Öffnen des Knotens, **ExpandAll** zum Öffnen aller Unterknoten und **Toggle** zum Umschalten zwischen geöffnetem und geschlossenem Zustand.

A.3 Menüs und Statusleisten

In jeder modernen Anwendung finden Sie im oberen Bereich immer eine Menüleiste und im unteren Bereich eine Statusleiste, die Ihnen Informationen, zum Beispiel über den Anmeldestatus, anzeigt. In diesem Abschnitt werden Sie die Steuerelemente zur Erzeugung von Menüleisten, Kontextmenüs und Statusleisten kennenlernen.

A.3.1 MenuStrip

Zur Erzeugung einer Menüleiste müssen Sie das Steuerelement **MenuStrip** verwenden.



Abb. A.27: Das Steuerelement `MenuStrip`

Zur Definition der einzelnen Menüpunkte müssen Sie der Eigenschaft **Items** einen Menüpunkt hinzufügen, der vom Typ **ToolStripMenuItem** ist. Hierzu können Sie innerhalb des Editors einfach in das Feld **HIER EINGEBEN** klicken und einen Text eintragen. Daraufhin werden automatisch weitere Menüeinträge hinzugefügt. Übernommen werden die Einträge jedoch immer erst bei Festlegung eines Textes. Die beiden anderen Möglichkeiten bestehen in der Definition der Einträge im Programmcode oder über einen Editor, den Sie über den **Button** im **EIGENSCHAFTENFENSTER** für die Eigenschaft **ITEMS** öffnen können. Wollen Sie den Menüeinträgen Bilder zuordnen, so müssen Sie dies über die Eigenschaft **Image** der

Klasse **ToolStripMenuItem** festlegen. Den Anzeigetext bestimmen Sie über die Eigenschaft **Text**. Über die Eigenschaft **ShortcutKeys** können Sie jedem Eintrag eine Tastenkombination zuweisen. Um zu ermitteln, welcher Eintrag ausgewählt wurde, können Sie sich für das Ereignis **ItemClicked** oder direkt für jedes **ToolStripMenuItem** für dessen Ereignis **Click** registrieren.

A.3.2 ToolStrip

Hierbei handelt es sich um eine einfache Statusleiste, mit der Sie zusätzliche Informationen anzeigen können. Die wichtigste Eigenschaft ist auch hier die Eigenschaft **Items**, mit der Sie Einträge hinzufügen können. Dabei haben Sie die Auswahl zwischen der Definition eines Labels (**ToolStripStatusLabel**), einer ProgressBar (**ToolStripProgressBar**), eines DropDownButtons (**ToolStripDropDownButton**) und eines SplitButtons (**ToolStripSplitButton**). Die folgende Abbildung zeigt die entsprechende Statusleiste mit den in der Reihenfolge genannten Komponenten:



Abb. A.28: Eine einfache Statusleiste

Diese Einträge können Sie entweder im Programmcode, über einen Editor oder im Designer selbst hinzufügen.

A.3.3 ToolStrip

In Visual Studio sehen Sie direkt unterhalb des Menüs eine Werkzeugliste. Diese wird mithilfe des Steuerelements **ToolStrip** erzeugt. Das Vorgehen entspricht dem des StatusStrip. Auch hier können Sie auf unterschiedlichste Weise über die Eigenschaft **Items** neue Einträge von folgenden Typen definieren:

- **ToolStripButton**
- **ToolStripLabel**
- **ToolStripSplitButton**
- **ToolStripDropDownButton**
- **ToolStripSeparator**
- **ToolStripComboBox**
- **ToolStripTextBox**
- **ToolStripProgressBar**

Die Komponenten weisen grundlegend dieselben Eigenschaften wie ihre Basispendants auf. Die folgende Abbildung zeigt das **ToolStrip** mit den o.g. Komponenten in der entsprechenden Reihenfolge:



Abb. A.29: Das ToolStrip

Über die Eigenschaft **GripStyle** können Sie zusätzlich festlegen, ob das Grip (zum Verschieben des ToolStrips) sichtbar ist.

A.3.4 ToolStripContainer

Jetzt, nachdem Sie sowohl eine Menü- als auch eine Statusleiste festlegen können, wird es Zeit, diese sinnvoll in Form eines Layoutcontainers zu platzieren (das Thema Layout wird in Abschnitt A.4. besprochen). Dazu eignet sich der **ToolStripContainer**, der es erlaubt, Steuerelemente auf dem Formular links, oben, rechts und unten anzudocken. Jeder Bereich wird hierbei durch ein **ToolStripPanel** festgelegt. Wenn Sie ein Panel belegen möchten, müssen Sie zuerst im Designer den entsprechenden Bereich über die Pfeilschaltflächen einblenden. Anschließend können Sie das Steuerelement darauf ziehen. Die wichtigste Eigenschaft zur Positionierung der Elemente ist die Eigenschaft **Dock** (auch diese wird in Abschnitt A.4 erörtert). Die einzelnen Panels lassen sich über die Eigenschaften **BottomToolStripPanel** (unten), **LeftToolStripPanel** (links), **RightToolStripPanel** (rechts), **TopToolStripPanel** (oben) und **ContentPanel** (Mitte) definieren. Die folgende Abbildung zeigt den Container im Designer:

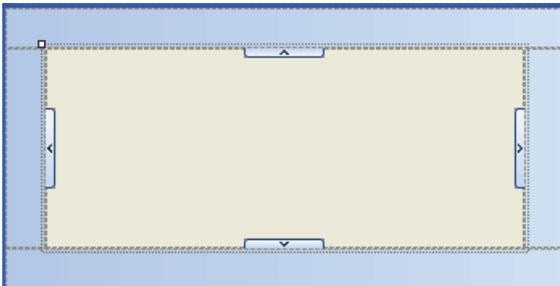


Abb. A.30: Der ToolStripContainer zur dynamischen Positionierung von Menü- und Statusleisten

A.3.5 ContextMenuStrip

Jedes Steuerelement besitzt die Eigenschaft **ContextMenuStrip**, der Sie eine Instanz des Steuerelements **ContextMenuStrip** zuweisen können. Ein **ContextMenuStrip** definiert ein Kontextmenü, das Sie mittels Rechtsklick auf ein Steuerelement öffnen können. Um ein solches Kontextmenü zu definieren, können Sie das Steuerelement zunächst aus der TOOLBOX auf das Formular ziehen. Anschließend definieren Sie wie beim **MenuStrip** die einzelnen Einträge und weisen das Steuerelement über dessen Eigenschaft im EIGENSCHAFTENFENSTER oder direkt im Programmcode einem anderen Steuerelement zu.

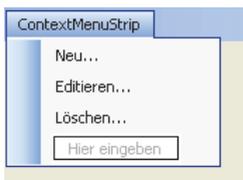


Abb. A.31: Das Kontextmenü im Designer

Jedes Element ist vom Typ **ToolStripMenuItem** und besitzt die Eigenschaft **Text** zur Definition des Anzeigetextes sowie die Eigenschaft **DropDownItems**, mit der dem aktuellen Ein-

trag weitere Untereinträge hinzugefügt werden können. Zusätzlich können Sie auch folgende Steuerelemente für die Einträge verwenden:

- **ToolStripComboBox**
- **ToolStripSeparator**
- **ToolStripTextBox**

Als wichtigste Ereignisse gelten **Click**, mit dem Sie auf das Klickereignis eines Eintrags reagieren können, und **ItemClicked**, mit dem Sie den angeklickten Eintrag direkt ermitteln können.

Das folgende Beispiel zeigt einen **Button**, dem ein Kontextmenü mit den Einträgen **NEU**, **EDITIEREN** und **LÖSCHEN** zugewiesen ist. Wenn auf eines der Elemente geklickt wird, soll dessen Bezeichnung auf dem **Button** angezeigt werden:

```
...  
private void ContextMenu_EntryClicked(object sender,  
ToolStripItemClickedEventArgs e)  
{  
    button1.Text = e.ClickedItem.Text;  
}
```

Listing A.17: Auswahl des angeklickten Kontextmenüeintrags

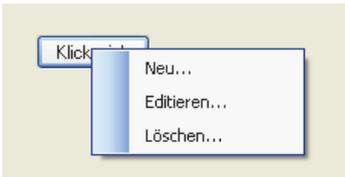


Abb. A.32: Das definierte Kontextmenü

A.4 Layout und Container

Wie bereits erwähnt, werden Steuerelemente mittels des Designers absolut positioniert. Jede Größenänderung des Fensters würde die Position und die Ausrichtung des Steuerelements nicht beeinflussen. Wenn Sie ein gewisses Layout definieren, wird die Anpassung durch das Layout automatisch vorgenommen. Zur Definition des Layouts existieren verschiedene Layoutcontainer, die in diesem Abschnitt näher betrachtet werden.

A.4.1 Die Eigenschaften Dock und Anchor

Zur wesentlichen Ausrichtung von Steuerelementen besitzt jedes Steuerelement die Eigenschaften **Dock** und **Anchor**. Diese steuern die Ausrichtung der Steuerelemente, wenn die Fenstergröße verändert wird.

Die Eigenschaft **Dock** ermöglicht es, das Steuerelement an die Außenkanten des übergeordneten Containers anzudocken. Dabei wird die Größe des Steuerelements nur dahingehend verändert, dass die Ausrichtung an den Kanten erhalten bleibt. Die folgende Abbildung zeigt die Auswirkungen der Eigenschaft auf ein Steuerelement:

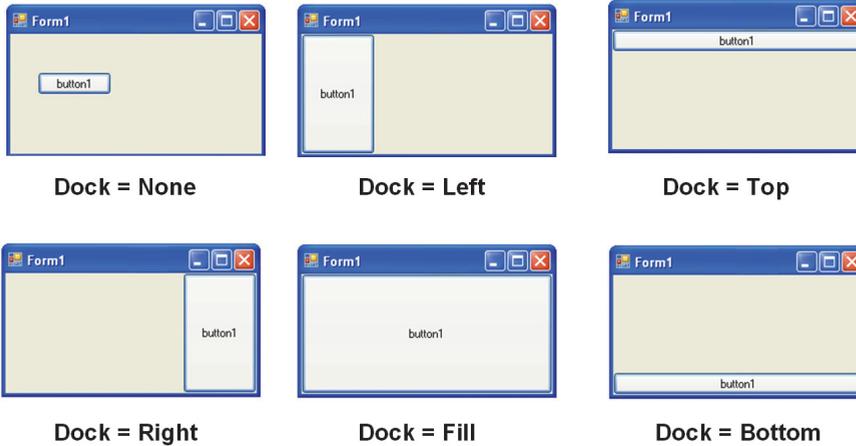


Abb. A.33: Die Eigenschaft Dock im Überblick

Eine weitere Eigenschaft zur Ausrichtung stellt die Eigenschaft **Anchor** dar. Mithilfe dieser Eigenschaft werden Steuerelemente relativ zur Außenkante des übergeordneten Containers ausgerichtet. Dies bedeutet, dass Sie hiermit den Abstand zu den Außenkanten bestimmen. Standardmäßig ist für jedes Steuerelement der Wert **Left, Top** vordefiniert. Sie können jedoch auch weitere Abstände definieren. Die folgende Abbildung zeigt die Auswirkungen der Eigenschaft auf ein Steuerelement:

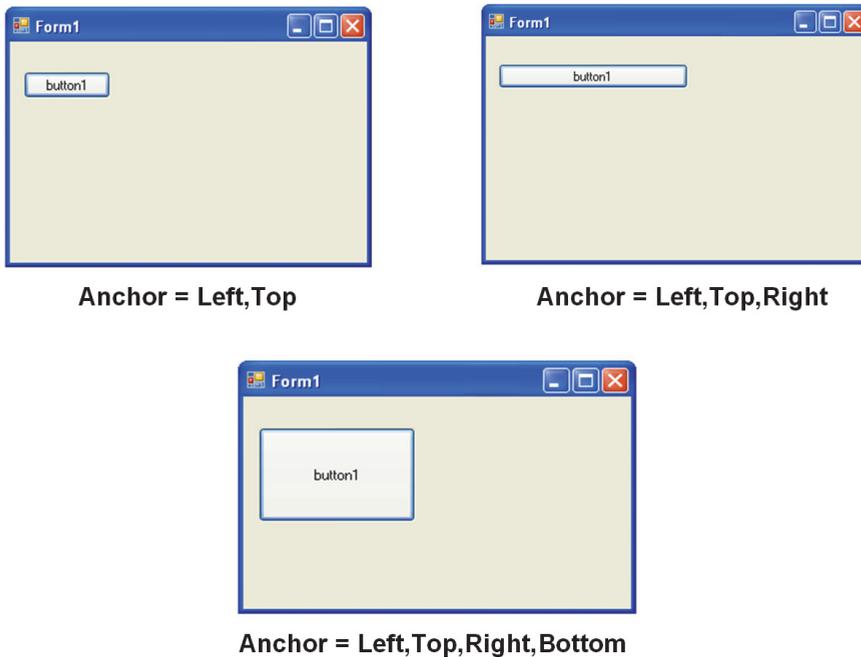


Abb. A.34: Die Eigenschaft Anchor in der Übersicht beim Vergrößern des Fensters

A.4.2 FlowLayoutPanel

Das **FlowLayoutPanel** ordnet die Steuerelemente dynamisch horizontal und vertikal aus. Wenn das Fenster vergrößert wird, werden die Steuerelemente wie Text in Word automatisch umbrochen, sofern sie durch die Größenänderung nicht mehr in die Zeile passen. Hierfür ist die Eigenschaft **WrapContents** verantwortlich, die standardmäßig auf den Wert **true** gesetzt ist. Die folgende Abbildung zeigt den Layoutcontainer in Aktion:



Abb. A.35: Das FlowLayoutPanel

A.4.3 GroupBox

Möchten Sie mehrere Steuerelemente zu einer Gruppe zusammenfassen, so müssen Sie das Steuerelement **GroupBox** verwenden. Um der **GroupBox** Elemente hinzuzufügen, besitzt sie wie das Steuerelement **Panel** die Eigenschaft **Controls**, mit der Sie über die Methode **Add** neue Steuerelemente hinzufügen können. Sie sollten also zur Zusammenfassung von Steuerelementen innerhalb einer **GroupBox** die Steuerelemente im Programmcode ergänzen:

```
...  
public Form1()  
{  
    InitializeComponent();  
    //Eine CheckBox und einen RadioButton gruppieren  
    groupBox1.Controls.Add(new CheckBox(){Location = new Point(20,30),  
    Text = "Option 1"});  
    groupBox1.Controls.Add(new RadioButton(){Checked = true,  
    Text = "Option 2", Location = new Point(20,50)});  
}
```

Listing A.18: Hinzufügen von Steuerelementen zur GroupBox im Programmcode

Das Ergebnis sieht folgendermaßen aus:



Abb. A.36: Der Container GroupBox

Weiterhin verfügt die **GroupBox** über die Eigenschaft **Text**, mit der sich der Titel der **GroupBox** festlegen lässt.

Vorsicht

Bedenken Sie, dass die Reihenfolge der Zuweisung für das Zeichnen der Steuerelemente wichtig ist. Sie sollten also zuerst die **GroupBox** und dann die Steuerelemente anlegen.

A.4.4 Panel

Das **Panel** ist eine einfache Komponente, auf der sich weitere Container platzieren lassen. Hierzu definiert der Container eine Eigenschaft **Controls**, mit der Elemente hinzugefügt werden können. Das Hinzufügen sollte dann wie bei der **GroupBox** im Code erfolgen. Die folgende Abbildung zeigt das **Panel**:

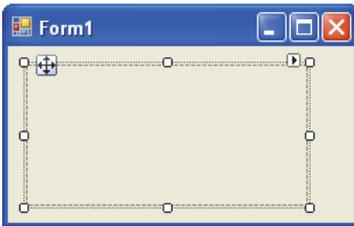


Abb. A.37: Das Panel

Wichtig

Wie bei der **GroupBox** kommt es auch beim **Panel** auf die Reihenfolge an.

A.4.5 SplitContainer

Wollen Sie ein Fenster mittels eines Trennbalkens visuell aufteilen, dann sollten Sie den **SplitContainer** verwenden. Der **SplitContainer** besteht aus zwei **Panel**s mit einem **Splitter** als Trennbalken. Ein einfaches Beispiel für den Container stellt die Aufteilung eines Fensters in die Bereiche Baumansicht links und dynamischer Inhalt rechts dar. Der Layoutcontainer **SplitContainer** verfügt über die folgenden wichtigen Eigenschaften:

- **Panel1**: Festlegung eines Layoutcontainers für den linken Bereich des Trennbalkens
- **Panel2**: Festlegung eines Layoutcontainers für den rechten Bereich des Trennbalkens
- **FixedPanel**: Legt fest, bei welchem Panel die Größe nach der Größenanpassung des Fensters beibehalten wird. Der Standardwert ist **None** (kein Panel).
- **IsSplitterFixed**: Legt fest, ob der Balken fest oder beweglich ist.
- **SplitterDistance**: Legt den Abstand des Trennbalkens von der linken bzw. oberen Kante fest.
- **SplitterIncrement**: Legt die Anzahl der Pixel fest, um die sich der Trennbalken schrittweise verschiebt. Der Standardwert ist 1.

- **SplitterWidth**: Legt die Breite des Trennbalkens fest.
- **Orientation**: Legt fest, ob der Trennbalken vertikal (**Vertical**) oder horizontal (**Horizontal**) ausgerichtet ist.
- **BorderStyle**: Legt den Rahmen fest. Ist der Standardwert **None** gesetzt, wird der Trennbalken nicht angezeigt.

Die folgende Abbildung zeigt den **SplitContainer** im Einsatz:



Abb. A.38: Der Layoutcontainer **SplitContainer** mit einem Baum links und dem dynamischen Inhalt rechts

A.4.6 TabControl

Das Steuerelement **TabControl** ermöglicht die Anordnung von Steuerelementen und Panels über Registerseiten:

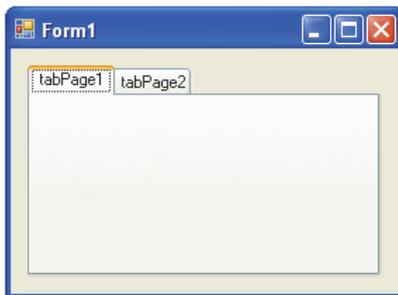


Abb. A.39: Das **TabControl** im Einsatz

Um neue Registerseiten hinzuzufügen oder bestehende zu löschen, gibt es drei Möglichkeiten:

1. **Verwendung des Smarttags**
2. **Verwendung des Editors**

Wenn Sie im **EIGENSCHAFTENFENSTER** auf die Eigenschaft **TABPAGES** klicken, erscheint ein Editor, mit dem Sie Registerseiten hinzufügen, entfernen und deren Eigenschaften individuell festlegen können:

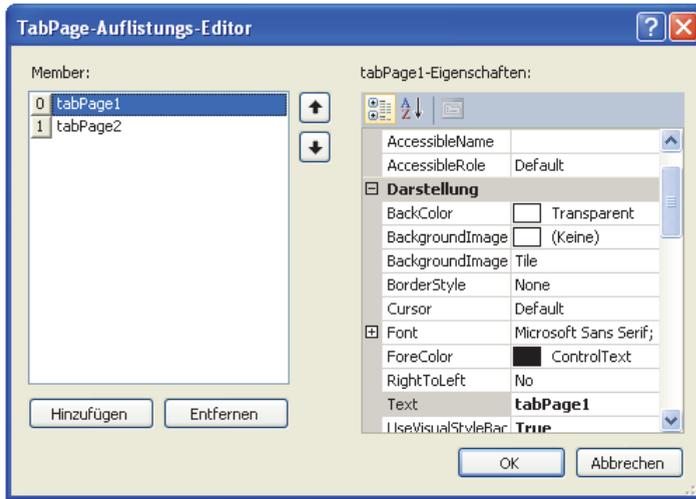


Abb. A.40: Der Editor für Registerseiten

3. Festlegung im Programmcode

Um innerhalb des Programmcodes eine neue Seite hinzuzufügen, müssen Sie der Eigenschaft **Controls** des **TabControl**s eine Instanz der Klasse **TabPage** hinzufügen:

```
...
public Form1()
{
    InitializeComponent();
    TabPage page = new TabPage("Neue Seite");
    tabControl1.Controls.Add(page);
}

```

Listing A.19: Hinzufügen einer neuen Seite im Programmcode

Weiterhin verfügt das **TabControl** über die folgenden wichtigen Eigenschaften und Ereignisse:

Eigenschaft/Ereignis	Bedeutung
ImageList	Legt die Liste an Bildern für die Registerseiten fest.
HotTrack	Legt fest, ob sich die visuelle Darstellung der Seiten ändert, wenn die Maus über den Seiten bewegt wird.
ItemSize	Legt die Größe der Seiten fest.
Multiline	Legt fest, ob die Seiten mehrzeilig sein können.
TabPages	Legt die Seiten für das TabControl fest.
SelectedTab	Gibt die ausgewählte Seite zurück.
RowCount	Ermittelt die Anzahl von Zeilen für eine Seite.

Tabelle A.33: Die wichtigsten Eigenschaften und Ereignisse des **TabControl**s

Eigenschaft/Ereignis	Bedeutung
Appearance	<p>Legt die Darstellungsform des TabControls fest. Diese ist vom Typ der Enumeration TabAppearance und definiert die folgenden Members:</p> <ul style="list-style-type: none"> ■ Buttons:  ■ FlatButtons:  ■ Normal: Die Standarddarstellung
Deselected	Dieses Ereignis tritt ein, wenn die Auswahl einer Seite aufgehoben wurde. Die entsprechend betroffene Seite ermitteln Sie über die Eigenschaft TabPage des Ereignisparameters TabControlEventArgs .
Deselecting	Dieses Ereignis tritt ein, wenn die Auswahl einer Seite aufgehoben wird. Die entsprechend betroffene Seite ermitteln Sie über die Eigenschaft TabPage des Ereignisparameters TabControlEventArgs .
Selected	Dieses Ereignis tritt ein, wenn eine Seite ausgewählt wurde. Die entsprechend betroffene Seite ermitteln Sie über die Eigenschaft TabPage des Ereignisparameters TabControlEventArgs .
Selecting	Dieses Ereignis tritt ein, wenn eine Seite gerade ausgewählt wird. Die entsprechend betroffene Seite ermitteln Sie über die Eigenschaft TabPage des Ereignisparameters TabControlEventArgs .

Tabelle A.33: Die wichtigsten Eigenschaften und Ereignisse des **TabControl**s (Forts.)

Die Reihenfolge der Ereignisse für die Auswahl und Abwahl von Registerseiten ist in der folgenden Abbildung dargestellt:



Abb. A.41: Der Ablauf von Ereignissen beim Wechseln einer Seite

A.4.7 TableLayoutPanel

Sollen die Steuerelemente dynamisch und tabellarisch angeordnet werden, empfiehlt sich die Verwendung des Layouts **TableLayoutPanel**. Dieses legt die Steuerelemente in Form eines Gitters in Spalten und Zeilen ab. Sie können die Anzahl der Zeilen und Spalten sowie deren Breite und Höhe individuell mittels eigenen Konfigurationsdialogs festlegen. Dieser Dialog lässt sich über den kleinen Pfeil, auch **Smarttag** genannt, aufrufen. Zugriff auf die einzelnen Spalten und Zeilen erhalten Sie über die Eigenschaften **Rows** für Zeilen und **Columns** für Spalten. Weiterhin lässt sich die Anzahl der Zeilen über die Eigenschaft **Row-**

Count und die Anzahl der Spalten über die Eigenschaft **ColumnCount** festlegen. Die folgende Abbildung zeigt die Verwendung des **TableLayoutPanel**s mit mehreren Buttons:

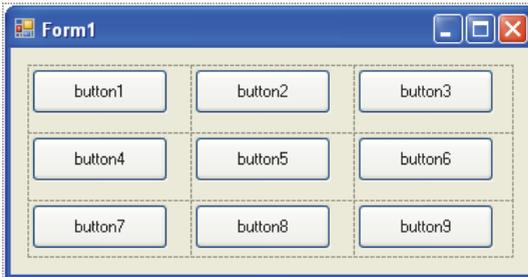


Abb. A.42: Das TableLayoutPanel

A.5 Allgemeine Dialoge

Für die Festlegung von Farben und Schriften sowie zur Auswahl von Pfaden existieren bereits fertige Dialoge, die Sie wie ein neues Formular über die Methode **ShowDialog** aufrufen können. Anschließend können Sie die das Ergebnis vom Typ **DialogResult** wie gewohnt auswerten. Im folgenden Abschnitt werden die entsprechenden Dialoge kurz vorgestellt.

A.5.1 ColorDialog

Der **ColorDialog** ist ein Dialog zur Festlegung von Farben, wie Sie ihn aus MICROSOFT PAINT kennen.



Abb. A.43: Der ColorDialog

Die wichtigste Eigenschaft ist die Eigenschaft **Color** als ARGB-Angabe, mit der Sie sowohl die ausgewählte Farbe vor der Anzeige des Dialogs festlegen als auch die ausgewählte Farbe

nach der Anzeige des Dialogs ermitteln können. Die weiteren Eigenschaften des Dialogs sind in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
AllowFullOpen	Legt fest, ob der erweiterte Dialog zur Definition eigener Farben eingeblendet werden soll.
FullOpen	Legt fest, ob beim Öffnen des Dialogs der erweiterte Dialog zur Definition eigener Farben eingeblendet werden soll.
AnyColor	Legt fest, ob alle zur Verfügung stehenden Farben zur Auswahl stehen.
CustomColors	Definiert eine Reihe von benutzerdefinierten Farben bzw. ermöglicht es, diese abzufragen.
SolidColorOnly	Legt fest, ob nur Volltonfarben ausgewählt werden können.
ShowHelp	Legt fest, ob eine Hilfeschnittfläche angezeigt werden soll.

Tabelle A.34: Die wichtigsten Eigenschaften des ColorDialogs

Im folgenden Beispiel werden benutzerdefinierte Farben vor der Anzeige des Dialogs definiert und anschließend wird die gewählte Farbe nach dem Schließen des Dialogs über eine MessageBox ausgegeben:

```
...  
colorDialog1.CustomColors = new int[]{};  
if(colorDialog1.ShowDialog() == DialogResult.OK)  
    MessageBox.Show("Gewählte Farbe: " + colorDialog1.Color.ToString());
```

Listing A.20: Festlegung von benutzerdefinierten Farben

A.5.2 FontDialog

Zur Auswahl von Schriftarten und deren Formatierung können Sie den **FontDialog** verwenden.



Abb. A.44: Der FontDialog

Ähnlich wie bei `ColorDialog` `Color` die wichtigste Eigenschaft zur Definition der ausgewählten Farbe ist, gibt es für den `FontDialog` die Eigenschaft `Font`, mit der Sie die ausgewählte Schrift sowohl vor dem Anzeigen des Dialogs als auch nach dem Schließen des Dialogs festlegen bzw. abfragen können. Da grundlegend jedes Steuerelement eine Eigenschaft `Font` besitzt, können Sie die ausgewählte Schrift über diese Eigenschaft einfach folgendermaßen zuweisen:

```
textBox1.Font = fontDialog1.Font;
```

Weitere Eigenschaften entnehmen Sie der folgenden Tabelle:

Eigenschaft	Bedeutung
<code>AllowScriptChange</code>	Legt fest, ob das Skript (Westlich, Türkisch etc.) geändert werden kann.
<code>AllowSimulations</code>	Legt fest, ob Windows die Schriften über GDI simulieren kann.
<code>AllowVectorFonts</code>	Legt fest, ob Vektorschriftarten ausgewählt werden können.
<code>AllowVerticalFonts</code>	Legt fest, ob vertikale Fonts möglich sind.
<code>Color</code>	Legt die aktuelle Schriftfarbe fest.
<code>FixedPitchOnly</code>	Legt fest, ob nur Schriftarten mit fester Zeichenbreite erlaubt sind.
<code>FontMustExist</code>	Legt fest, ob nur vorhandene Schriftarten ausgewählt werden können.
<code>MaxSize</code>	Legt den maximalen Grad fest, der ausgewählt werden kann.
<code>MinSize</code>	Legt den minimalen Grad fest, der ausgewählt werden kann.
<code>ShowColor</code>	Legt fest, ob die <code>ComboBox</code> zur Farbauswahl zusätzlich angezeigt werden soll.
<code>ShowEffects</code>	Legt fest, ob Effekte wie Unterstreichen, Farbe und Durchstreichen angezeigt werden sollen.
<code>ScriptsOnly</code>	Legt fest, ob Skripte erlaubt sind.
<code>ShowApply</code>	Legt fest, ob die Schaltfläche ÜBERNEHMEN angezeigt werden soll.
<code>ShowHelp</code>	Legt fest, ob die Hilfeschaltfläche angezeigt werden soll.

Tabelle A.35: Die wichtigsten Eigenschaften des `FontDialogs`

A.5.3 FolderBrowserDialog

Der `FolderBrowserDialog` entspricht der Funktionsweise des `OpenFileDialogs`, nur dass Sie statt Dateien Ordner auswählen können.



Abb. A.45: Der FolderBrowserDialog

Die wichtigste Eigenschaft dieser Komponente ist die Eigenschaft **SelectedPath**, mit der Sie vor dem Anzeigen und nach dem Schließen des Dialogs den ausgewählten Pfad festlegen bzw. auslesen können. Weitere wichtige Eigenschaften sind in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
Description	Legt den Text fest, der über der Baumstruktur angezeigt werden soll.
RootFolder	Legt den Ordner fest, der als Startpunkt für die Suche verwendet wird. Es werden nur der Ordner und alle untergeordneten Ordner angezeigt.
ShowNewFolderButton	Legt fest, ob die Schaltfläche NEUER ORDNER angezeigt wird.

Tabelle A.36: Die wichtigsten Eigenschaften des FolderBrowserDialogs

A.6 Drucken

Das Drucken aus Anwendungen heraus gehört heutzutage zu den wichtigsten Funktionen überhaupt. In diesem Abschnitt werden Sie die hierfür benötigten Dialoge und Steuerelemente kennenlernen. Es sollen jedoch lediglich die Grundlagen vermittelt werden. Für eine umfangreiche Einweisung in die Thematik sei auf entsprechende Fachliteratur verwiesen.

A.6.1 PrintDocument

Die Komponente **PrintDocument** ist die zentrale Komponente zur Festlegung von Druckereinstellungen. Das wichtigste Ereignis ist das Ereignis **PrintPage**, das dann ausgelöst wird, wenn das Dokument gedruckt werden soll. Über den Ereignisparameter erhalten Sie ein **Graphics**-Objekt (mehr dazu im nächsten Kapitel zu GDI+), mit dem Sie den zu drucken-

den Inhalt bestimmen können. Zum Auslösen des Druckvorgangs müssen Sie die Methode **Print** verwenden. Wollen Sie zusätzlich den Namen des Dokuments festlegen, können Sie die Eigenschaft **DocumentName** verwenden.

Druckeinstellungen festlegen

Die Druckereigenschaften werden über die Eigenschaft **PrinterSettings** festgelegt. Dies stellt die Klasse **PrinterSettings** dar, die die folgenden Eigenschaften definiert:

Eigenschaft	Bedeutung
CanDuplex	Bestimmt, ob der Drucker beidseitiges Drucken unterstützt.
Collate	Legt fest, ob das Dokument sortiert wird.
Copies	Legt die Anzahl der Kopien des Dokuments fest.
DefaultPageSettings	Bestimmt die Standarddruckereinstellungen.
Duplex	Bestimmt die Einstellungen für beidseitiges Drucken.
FromPage	Bestimmt die Nummer der ersten zu druckenden Seite.
IsDefaultPrinter	Bestimmt, ob es sich um den Standarddrucker handelt.
PaperSizes	Bestimmt die unterstützten Papiergrößen.
PaperSources	Bestimmt die verfügbaren Papierzufuhrschächte.
PrinterName	Bestimmt den Druckernamen.
PrinterResolutions	Bestimmt die Auflösung des Druckers.
PrintRange	Bestimmt die Seitennummern der zu druckenden Seiten.
SupportsColor	Bestimmt, ob der Drucker Farbdruck unterstützt.
ToPage	Bestimmt die Nummer der letzten zu druckenden Seite.

Tabelle A.37: Die wichtigsten Eigenschaften der Klasse PrinterSettings

Das folgende Beispiel definiert als Druckerausgabe einen Text:

```
...
private void PrintButton_Click(object sender, EventArgs e)
{
    //Startet den Druckvorgang
    printDocument1.DocumentName = "Testdruck";
    printDocument1.Print();
}

private void PrintPage(object sender, PrintPageEventArgs e)
{
    //Ausgabe eines Textes
    e.Graphics.DrawString("Hello World wird gedruckt",
        new Font("Arial", 14.0F), Brushes.Black, 100,100);
}
}
```

Listing A.21: Beispielcode für das Drucken unter Windows Forms

Hinweis

Das Graphics-Objekt verfügt über die Eigenschaft **PageUnit**, mit der Sie die zu verwendende Maßeinheit angeben können. Diese Eigenschaft ist vom Typ **GraphicsUnit** und definiert als Enumeration die folgenden Members:

- **Millimeter**: Legt Millimeter als Maßeinheit fest.
- **Document**: Legt die Dokumenteneinheit (1/300 Zoll) als Maßeinheit fest.
- **Pixel**: Legt die Gerätepixel als Maßeinheit fest.
- **Point**: Legt den Druckerpunkt (1/72 Zoll) als Maßeinheit fest.
- **World**: Legt das globale Koordinatensystem als Maßeinheit fest.
- **Inch**: Legt Zoll als Maßeinheit fest.
- **Display**: Legt die Maßeinheit des Anzeigergerätes als Maßeinheit fest.

A.6.2 PrintDialog

Um den Ihnen bekannten Druckdialog anzuzeigen, müssen Sie die Komponente **PrintDialog** verwenden.

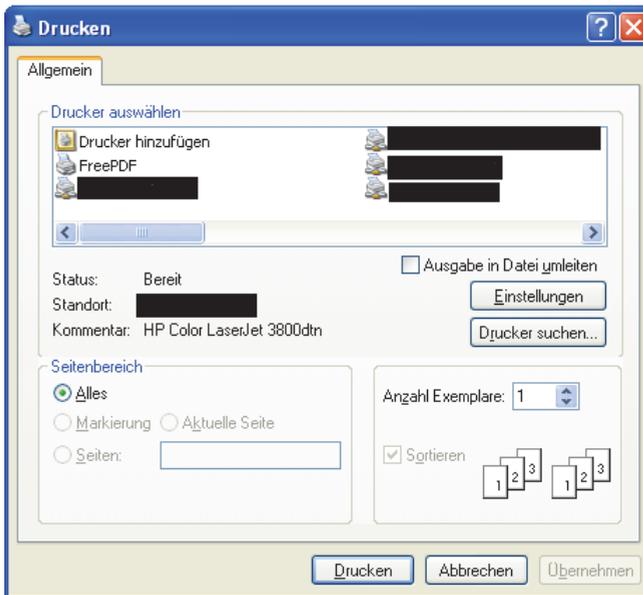


Abb. A.46: Der PrintDialog

Über die Eigenschaft **Document** können Sie ein bestehendes PrintDocument-Objekt übergeben, das automatisch die Einstellungen des **PrintDialogs** übernimmt. Weitere Eigenschaften des **PrintDialogs** sind in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
AllowPrintToFile	Legt fest, ob das Kästchen AUSGABE IN DATEI aktiviert ist.
AllowCurrentPage	Legt fest, ob die Schaltfläche AKTUELLE SEITE angezeigt wird.
AllowSelection	Legt fest, ob die Schaltfläche AUSWAHL aktiviert ist.
AllowSomePages	Legt fest, ob die Schaltfläche SEITEN angezeigt wird.
ShowHelp	Legt fest, ob die Hilfeschnittfläche angezeigt werden soll.
PrinterSettings	Bestimmt die Druckereinstellungen für diesen Dialog.
PrintToFile	Bestimmt den Wert des Kästchens AUSGABE IN DATEI.

Tabelle A.38: Die wichtigsten Eigenschaften des PrintDialogs

Das folgende Beispiel erweitert das Beispiel aus Listing A.21 um die Komponente:

```

...
private void PrintButton_Click(object sender, EventArgs e)
{
    printDialog1.Document = printDocument1;
    printDocument1.DocumentName = "Testdruck";

    //Drucken des Dokuments
    if(printDialog1.ShowDialogResult() == DialogResult.OK)
        printDocument1.Print();
}

```

Listing A.22: Anzeige des Druckdialogs zur Übernahme von Druckereinstellungen

A.6.3 PageSetupDialog

Den **PageSetupDialog** kennen Sie bereits. Er erscheint, wenn Sie beispielsweise das EIGENSCHAFTENFENSTER für den ausgewählten Drucker aufrufen:

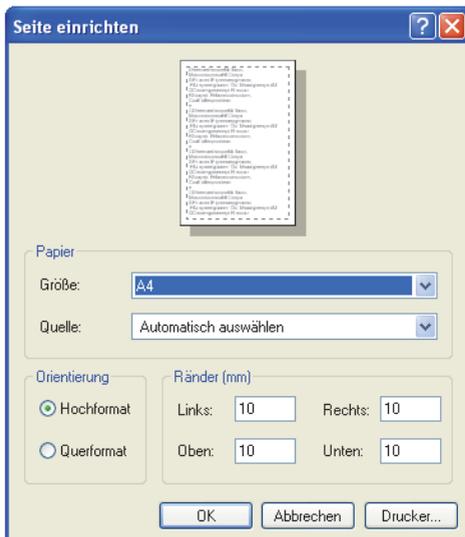


Abb. A.47: Der PageSetupDialog

Wollen Sie die Einstellungen des Dialogs für ein bestehendes Dokument übernehmen, können Sie hierzu die Eigenschaft **Document** verwenden. Dies entspricht dem Verhalten von `PrintDialog`. Die weiteren Eigenschaften des Dialogs sind in der folgenden Tabelle dargestellt:

Eigenschaft	Bedeutung
AllowMargins	Bestimmt, ob der Bereich SEITENRÄNDER aktiviert ist.
AllowOrientation	Bestimmt, ob der Bereich AUSRICHTUNG (horizontal und vertikal) aktiviert ist.
AllowPaper	Bestimmt, ob der Bereich PAPIER zur Festlegung der Papiergröße und der Papierzufuhr aktiviert ist.
AllowPrinter	Bestimmt, ob die Schaltfläche DRUCKER aktiviert ist.
MinMargins	Bestimmt die minimal auswählbare Randbreite in 1/100 Zoll.
PageSettings	Bestimmt die Seiteneinstellungen.
PrinterSettings	Legt die Druckereinstellungen fest.
ShowHelp	Legt fest, ob die Hilfeschaltfläche angezeigt wird.
ShowNetwork	Bestimmt, ob die Schaltfläche NETZWERK angezeigt wird.

Tabelle A.39: Die wichtigsten Eigenschaften der Komponente `PageSetupDialog`

A.6.4 PrintPreviewDialog

Wenn Sie eine Vorschau des Drucks anzeigen möchten, müssen Sie die Komponente **PrintPreviewDialog** verwenden. Um ein Dokument für die Vorschau zu definieren, können Sie der Eigenschaft **Document** eine Instanz des Typs `PrintDocument` zuweisen:

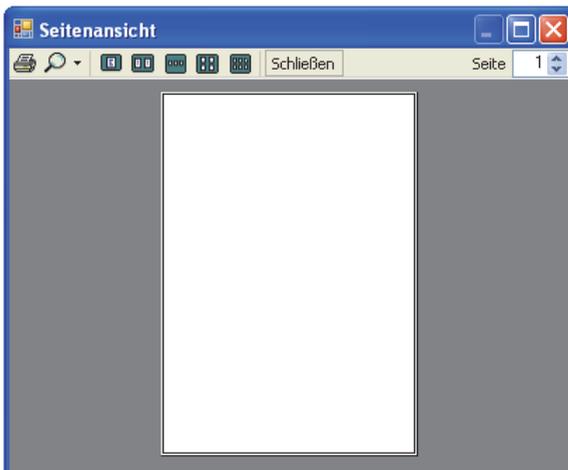
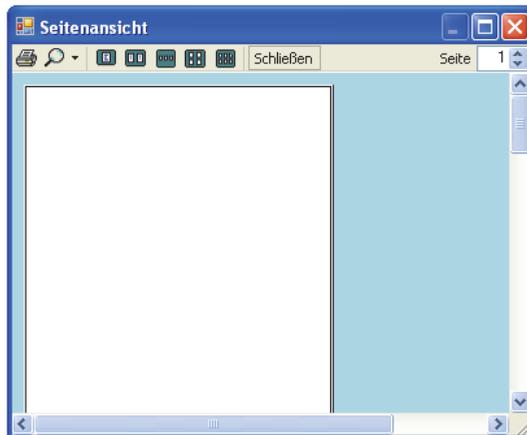


Abb. A.48: Der `PrintPreviewDialog`

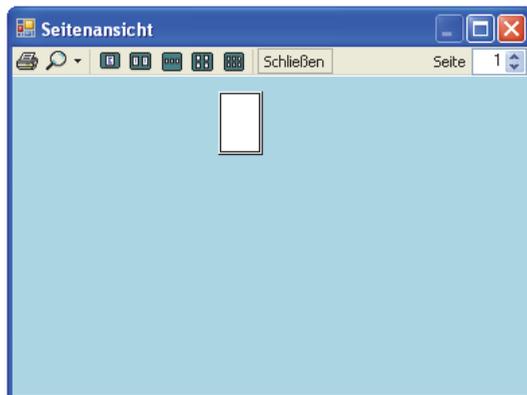
A.6.5 PrintPreviewControl

Die Ausgabe des `PrintPreviewDialogs` ist noch sehr simpel gehalten. Wollen Sie die Vorschau entsprechend Ihren Bedürfnissen anpassen, müssen Sie das Steuerelement **PrintPreviewControl** verwenden. Der `PrintPreviewDialog` definiert bereits eine Eigenschaft **PrintPreviewControl**, dessen Eigenschaften Sie nutzen können, um die Vorschau entsprechend anzupassen. Die hierfür wichtigen Eigenschaften sind in der folgenden Tabelle aufgelistet:

Eigenschaft	Bedeutung
AutoZoom	Legt fest, ob die Seiten so skaliert werden, dass alle Seiten flächenfüllend angezeigt werden: ■ AutoZoom = false:



■ **AutoZoom = true:**



UseAntialias	Nutzt das Antialiasing-Verfahren, um die Darstellungsqualität zu erhöhen.
---------------------	---

Tabelle A.40: Die wichtigsten Eigenschaften für das Steuerelement `PrintPreviewControl`

Eigenschaft	Bedeutung
Columns	Legt fest, wie viele Spalten und damit wie viele Seiten nebeneinander angezeigt werden können.
Document	Legt das zugehörige <code>PrintDocument</code> fest.
Rows	Legt fest, wie viele Zeilen und damit wie viele Seiten untereinander angezeigt werden können.
StartPage	Legt die Seitenzahl fest, die links oben angezeigt wird. Wenn Sie den Wert ändern, können Sie durch die weiteren Seiten navigieren und diese anzeigen lassen.
Text	Legt den zugehörigen Text fest.
Zoom	Legt den Zoomfaktor fest.
BackColor	Bestimmt die Hintergrundfarbe der Vorschau.

Tabelle A.40: Die wichtigsten Eigenschaften für das Steuerelement `PrintPreviewControl`

Das folgende Beispiel definiert `AutoZoom = false` für das Steuerelement und färbt den Hintergrund ein:

```
...  
printPreviewDialog1.PrintPreviewControl.BackColor = Color.LightBlue;  
printPreviewDialog1.PrintPreviewControl.AutoZoom = false;  
printPreviewDialog1.Document = printDocument1;  
printPreviewDialog1.PrintPreviewControl.Rows = 4;  
printPreviewDialog1.PrintPreviewControl.Columns = 2;
```

Listing A.23: Definition des `PrintPreviewControl`s

A.7 Benutzerdefinierte Steuerelemente

In manchen Situationen kommt es vor, dass Sie eine Gruppierung von Steuerelementen mehrmals in Form eines eigenständigen Steuerelements benötigen. Beispielsweise benötigen Sie eine Kombination aus `Label` und `TextBox`. Indem Sie diese Kombination in ein eigenes Steuerelement auslagern, minimiert sich der Aufwand für die Umsetzung an anderer Stelle innerhalb Ihrer Anwendung. Benutzersteuerelemente sind von der Basisklasse `UserControl` abgeleitet. Um ein Steuerelement zu erstellen, müssen Sie zunächst mittels Rechtsklick auf das Projekt und über `HINZUFÜGEN | BENUTZERSTEUERELEMENT` den Dialog zur Definition des Steuerelements öffnen. Anschließend öffnet sich der Dialog `NEUES ELEMENT HINZUFÜGEN`, in dem die Vorlage `BENUTZERSTEUERELEMENT` bereits ausgewählt ist. Geben Sie dem Steuerelement lediglich einen Namen. In dem vorliegenden Beispiel lautet der Name `INPUTCONTROL`. Im Designer werden Sie nun ein leeres `Panel` sehen. Jetzt können Sie dem `Panel` entweder mittels Designer oder im Programmcode Steuerelemente hinzufügen und diese über deren Eigenschaften konfigurieren. Ziehen Sie nun ein `Label` auf das `Panel` und geben Sie der Eigenschaft `Name` die Bezeichnung `LBLDESCRIPTION`. Auf dieselbe Weise verfahren Sie mit einer `TextBox` und geben ihr den Bezeichner `TXTBXINPUT`. Das Steuerelement sollte nun so aussehen:

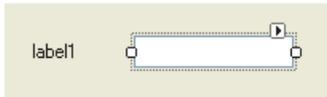


Abb. A.49: Ein benutzerdefiniertes Steuerelement

Da es sich um ein Steuerelement handelt, bei dem aus Gründen der Kapselung kein Zugriff von außen erfolgen soll, müssen Sie innerhalb des Steuerelements eigene Eigenschaften definieren. Im Folgenden wird nun eine Eigenschaft **Input** definiert, mit der die Eigenschaft **Text** der **TextBox** gesetzt und ausgelesen werden kann. Dasselbe wird mit dem **Label** durchgeführt, nur dass hier eine Eigenschaft **Description** erzeugt wird, über die der Text des **Labels** gesetzt werden kann:

```
...
public partial class InputControl : UserControl
{
    public InputControl()
    {
        InitializeComponent();
    }

    public string Input
    {
        set
        {
            txtBxInput.Text = value;
        }
        get
        {
            return txtBxInput.Text;
        }
    }

    public string Description
    {
        set
        {
            lblDescription.Text = value;
        }
    }
}
```

Listing A.24: Definition des Steuerelements `InputControl`

Innerhalb der **TOOLBOX** existiert der Reiter **ALLGEMEIN**. Ziehen Sie das Steuerelement aus dem **PROJEKTMAPPEN-EXPLORER** auf die freie Fläche des Reiters und erzeugen Sie das Projekt über das Menü unter **ERSTELLEN | PROJEKTMAPPE NEU ERSTELLEN**. Wenn das Projekt erfolgreich erstellt wurde, sehen Sie innerhalb der **Toolbox** einen neuen Reiter mit der Bezeichnung **[PROJEKT]KOMponenten**, der das Steuerelement beinhaltet. Nun können Sie das Steuerelement wie alle anderen Steuerelemente aus der **TOOLBOX** auf das Formular ziehen:

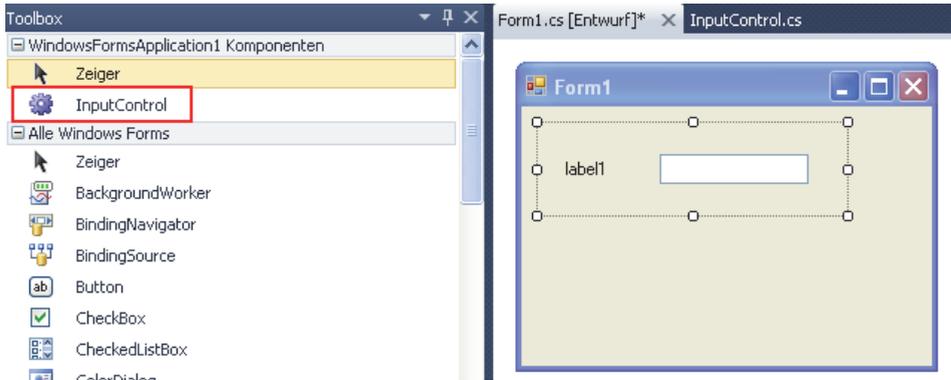


Abb. A.50: Wiederverwendung eines benutzerdefinierten Steuerelements in Visual Studio

Nun können Sie die Eigenschaften im Programmcode festlegen:

```
...  
public Form1()  
{  
    InitializeComponent();  
    //Initialisierung des Steuerelements  
    inputControl1.Description = "Eingabe:";  
    inputControl1.Input = "Hello World";  
}
```

Listing A.25: Festlegung der Eigenschaften des Steuerelements im Programmcode

Das Ergebnis sieht folgendermaßen aus:



Abb. A.51: Ergebnis des benutzerdefinierten Steuerelements

A.8 Einstellungen und Ressourcen

Gewissermaßen haben Sie bereits in Kapitel 9 zum Thema Dateien gelernt, dass Sie Einstellungen für eine Anwendung in einer Datei speichern und aus einer Datei laden sollten. Jedes Windows-Forms-Projekt besitzt eine Datei mit der Bezeichnung `SETTINGS.SETTINGS`, die sich im Projektordner `PROPERTIES` befindet. Innerhalb dieser Datei können Sie mittels Doppelklick auf die Datei bestimmte Werte definieren, die Sie beim Start auslesen können. Dies ist beispielsweise sinnvoll, wenn Sie den Text von Labels in einer Datei fest definieren und nicht im Programmcode fest verankern wollen. So können Sie jederzeit an einer zentralen Stelle Änderungen durchführen, ohne im Programmcode danach suchen zu müssen.

Name	Typ	Bereich	Wert
GreetingLabel	string	Benutzer	Hallo

Abb. A.52: Ausschnitt aus der Datei SETTINGS.SETTINGS

Um eine Eigenschaft zu definieren, geben Sie ihr zunächst über die Eigenschaft **NAME** eine Bezeichnung. Anschließend können Sie den **TYP** der Eigenschaft sowie den **ANWENDUNGSBEREICH** festlegen. Abschließend definieren Sie noch einen **WERT**. Dies alles wird in der Klasse **Settings** hinterlegt. Wollen Sie nun im Programmcode darauf zugreifen, müssen Sie zunächst eine **using**-Anweisung für den Namensraum **Properties** definieren und danach folgende Anweisung verwenden:

```
using MyApplication.Properties;
...
public Form1()
{
    InitializeComponent();
    //Es existiert ein Label, dem der definierte Text
    //zugewiesen werden soll
    label1.Text = Settings.Default.GreetingLabel;
}
```

Listing A.26: Verwendung von Eigenschaften der Klasse Settings

Auf die gleiche Weise können Sie auch Ressourcen wie Bilder und Icons innerhalb Ihrer Anwendung zuweisen. Zur Definition von Ressourcen existiert im selben Ordner die Datei **RESOURCES.RESX**. Wenn Sie auch hier einen Doppelklick ausführen, öffnet sich der Editor zur Definition von Ressourcen. Über das Menü können Sie zum einen den Typ der Ressource festlegen (**ZEICHENFOLGEN**, **BILDER**, etc.) und zum anderen auch bestimmen, ob eine neue oder bereits vorhandene Ressource hinzugefügt werden soll. Beim Hinzufügen einer neuen Ressource müssen Sie dieser zunächst einen Namen geben. Im vorliegenden Beispiel wird ein bestehendes Bild hinzugefügt und anschließend der Name des Bildes in **GREETINGS-IMAGE** geändert:

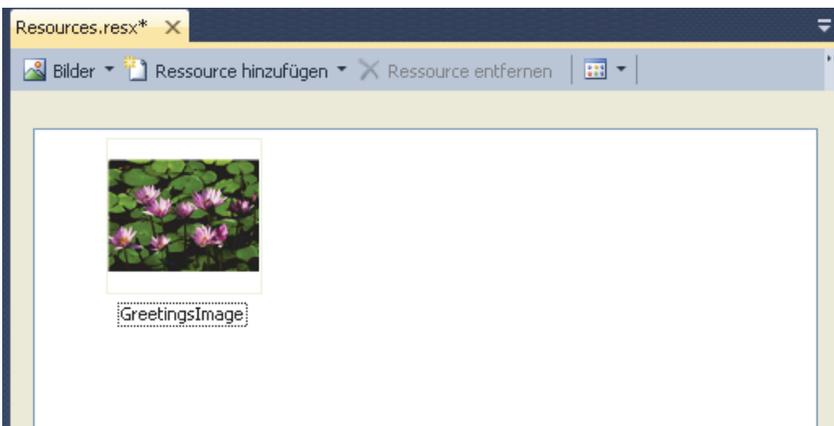


Abb. A.53: Das hinzugefügte Bild im Ressourcenverzeichnis

Wollen Sie das Bild verwenden, gehen Sie wie bei den Einstellungen vor, nur dass Sie aus demselben Namensraum statt der Klasse `Settings` die Klasse **Resources** verwenden:

```
using MyApplication.Properties;
...
public Form1()
{
    InitializeComponent();
    //Es existiert bereits ein PictureBox
    pictureBox1.Image = Resources.GreetingsImage;
}
```

Listing A.27: Zuweisung eines in den Ressourcen definierten Bildes im Programmcode

A.9 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie einfache grafische Oberflächen mit Windows Forms erzeugen können. Zunächst wurden der Aufbau eines Formulars und die Definition von Steuerelementen, Layouts und Ereignissen im Designer und im Programmcode betrachtet. Anschließend haben Sie die wichtigsten Steuerelemente und deren Eigenschaften sowie deren Verwendung kennengelernt. Weiterhin haben Sie gesehen, dass über Layoutcontainer die Positionierung von Steuerelementen dynamisch und leicht möglich ist. Darüber hinaus haben Sie einen ersten Einblick in die Thematik des Druckens unter Windows Forms gewonnen und einen ersten Vorgeschmack auf das Zeichnen mit GDI+ erhalten. Abschließend haben Sie gelernt, wie Sie mittels benutzerdefinierter Steuerelemente weitere Steuerelemente zu einem neuen Steuerelement zusammenfassen können. Dadurch minimieren Sie den späteren Aufwand und ermöglichen eine einfache Wiederverwendbarkeit.

Grafikprogrammierung mit GDI+

Nachdem Sie sich im ersten Teil des Anhangs intensiv mit der Oberflächenprogrammierung mit Windows Forms beschäftigt haben, werden Sie sich in diesem Teil mit der eigentlichen Grafikprogrammierung befassen. Hierzu wird die Grafikbibliothek **GDI+** (*Graphics Design Interface*) des .NET Frameworks verwendet. In diesem Kapitel werden Sie Folgendes lernen:

- Das Koordinatensystem
- Zeichnen von Linien, Rechtecken, Kreisen etc. mithilfe der Klasse `Graphics`
- Arbeiten mit Bildern
- Transformationen von Grafiken und Bildern
- Textausgabe
- Das Farbsystem
- Arbeiten mit Pinsel und Stift

Wichtig

Bedenken Sie, dass die Arbeit mit Grafiken im Zusammenhang mit vielen Ressourcen steht, die Gebrauch von der Schnittstelle `IDisposable` machen. Erschrecken Sie also nicht, falls Ihnen in den Codebeispielen oft ein `using`-Block begegnet.

B.1 Was ist GDI+?

GDI steht, wie bereits erwähnt, für Graphics Design Interface und stellt eine Bibliothek zur Grafikprogrammierung unter Windows zur Verfügung. GDI ist der Vorgänger der eigentlichen Bibliothek GDI+. Dabei ist GDI+ umfangreicher als das alte GDI, das auch heute noch in zahlreichen Win32-Anwendungen eingesetzt wird. Im Gegensatz zu GDI, bei dem jedes Grafikobjekt seine eigenen Zeichenmethoden definiert, gibt es in GDI+ lediglich eine zentrale Klasse, die allgemeine Methoden zum Zeichnen zur Verfügung stellt: die Klasse **Graphics**. Weiterhin müssen alle relevanten Informationen zum Zeichnen nicht mehr vor Beginn des Zeichenvorgangs definiert werden. Die für das Zeichnen notwendigen Informationen werden stets als Parameter der Methoden der Klasse **Graphics** übergeben. Darüber hinaus bietet GDI+ ...

- ... mehr unterschiedliche Farben und eine bessere Farbverwaltung
- ... verschiedene Bildformate, wie beispielsweise .BMP, .JPEG, .PNG
- ... Bildtransformationen
- ... Unterstützung von Antialiasing

- ... Unterstützung von Transparenz durch den Alpha-Kanal
- ... Farbverläufe

Wie Sie sehen, sind Ihnen für die Grafikprogrammierung also fast keine Grenzen gesetzt.

B.1.1 Namensräume und Klassen für die Grafikprogrammierung

Für die Grafikprogrammierung können Sie verschiedene Klassen aus dem Namensraumbereich **System.Drawing** verwenden. Zu diesem Namensraum gehören vor allem folgende Klassen:

- **Color**: Dient zur Transformation von Farben und definiert eigene Farbkonstanten.
- **Point**: Dient zur Definition eines Punktes im 2D-Raum, der durch eine x- und y-Koordinate definiert wird. Um einen Punkt mittels Gleitkommawert anzugeben, existiert die Klasse **PointF**, die die entsprechenden Koordinaten als Werte vom Typ `float` entgegennimmt.
- **Rectangle**: Dient zur Definition eines Rechtecks im 2D-Raum. Wollen Sie Gleitkommawerte verwenden, müssen Sie die Klasse **RectangleF** verwenden.
- **Size**: Dient zur Definition einer Größe im Sinne eines rechteckigen Bereichs. Dabei wird sowohl die Breite (**Width**), als auch die Höhe (**Height**) definiert.
- **Pen**: Stellt einen Zeichenstift dar. Diese Klasse wird in Abschnitt B.7.1 eingehender behandelt.
- **Brush**: Dient zur Definition von Pinselarten. Weitere mit **Brush** verwandte Klassen sind die Klassen **SolidBrush** und **TextureBrush**, die in Abschnitt B.7.2 erörtert werden.
- **Font**: Dient zur Definition von Schrifteigenschaften wie der Schriftart.
- **Bitmap**: Dient zur Definition und Verwaltung von Bitmaps.
- **Image**: Dient zur Definition und Verwaltung von Bildern allgemein.
- **Graphics**: Die zentrale Klasse zum Arbeiten mit Grafiken.

Ein weiterer Namensraum ist **System.Drawing.Drawing2D**, der Klassen zur Definition von Farbverläufen, Vektorgrafiken und Matrizen zur Transformation beinhaltet.

B.2 Das Koordinatensystem

Wie aus der Schulmathematik bekannt, existieren Punkte innerhalb eines Koordinatensystems. Entgegen dem uns bekannten System, ist beim Koordinatensystem in GDI+ der Nullpunkt allerdings nicht links unten, sondern oben links (siehe Abbildung B.1)

Die Einheit lautet **PIXEL**. Das Koordinatensystem ist an das des Bildschirms angepasst, bei dem dieselben Voraussetzungen und Definitionen gelten. Wichtig ist hierbei noch, dass sowohl die Y-Achse als auch die X-Achse in der dargestellten Form in einen positiven Raum verlaufen. Würden Sie das gezeigte Koordinatensystem an der X-Achse spiegeln, würden Sie wieder das Ihnen aus der Schulmathematik vertraute Koordinatensystem erhalten.

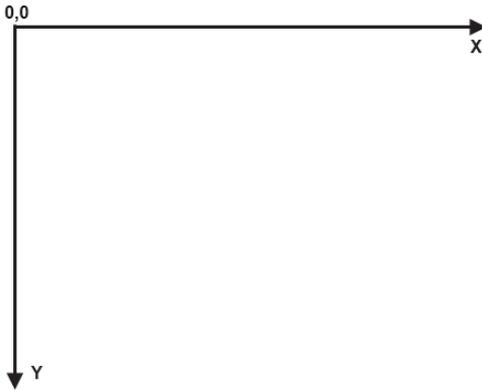


Abb. B.1: Das Koordinatensystem von GDI+

Weiterhin ist zu beachten, dass in GDI+ drei verschiedene Koordinatensysteme unterschieden werden:

- **Globale Koordinaten:** Dies stellt das ursprüngliche Koordinatensystem dar. Pixelkoordinaten werden dabei auf Gerätekoordinaten umgewandelt.
- **Seitenkoordinaten:** Hierbei wird der ursprüngliche Nullpunkt verschoben. Dies wird als **globale Transformation** bezeichnet. Zu den Transformationen gehören die **Translation** (Verschiebung), die **Skalierung** und die **Rotation**. Zu beachten ist, dass eine Translation und anschließende Rotation nicht dasselbe Ergebnis liefert wie eine Rotation und anschließende Translation. Deshalb ist die Reihenfolge der Transformationen besonders wichtig.
- **Gerätekoordinaten:** Dies stellt das Koordinatensystem des Ausgabegeräts wie Drucker oder Bildschirm dar. Da deren Auflösung in **Dpi** (*Dots per Inch*) angegeben wird und unterschiedlich sein kann, müssen Sie besonders aufpassen. Solange Sie nicht mit Gerätekoordinaten arbeiten, werden Sie für unterschiedliche Geräte stets unterschiedliche Ausgaben erhalten. Sie können natürlich auch gerätespezifisch arbeiten, indem Sie eine andere Maßeinheit wie Inch oder Millimeter wählen. Hierzu können Sie die Eigenschaft **PageUnit** verwenden.

B.2.1 Die Klasse Graphics

Wie bereits in diesem Kapitel erwähnt, stellt die Klasse **Graphics** die zentrale Klasse für die Grafikprogrammierung dar. Sie stellt verschiedene Methoden zum Zeichnen von Objekten und Formen zur Verfügung, die Sie im nächsten Abschnitt kennenlernen werden. Wichtig ist, dass Sie zum Zeichnen ein Formular benötigen. Die Klasse **Form** definiert das Ereignis **Paint**, das immer ausgelöst wird, wenn etwas gezeichnet werden soll. Dieses Ereignis liefert eine Instanz der Klasse **PaintEventArgs**, welche die Eigenschaft **Graphics** definiert und das mit dem Formular verbundene **Graphics**-Objekt liefert:

```
public partial class Form1 : Form
{
    public Form1()
    {
```

```

//Für das Ereignis Paint registrieren
this.Paint += FormPaint;
}

//Eventhandler des Ereignisses Paint
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = e.Graphics)
    {
        //Zeichnen
    }
}
}

```

Listing B.1: Verwendung des Graphics-Objekts innerhalb des Eventhandlers des Ereignisses Paint

Eine andere Möglichkeit zur Erzeugung eines **Graphics**-Objekts ist die Verwendung der Methode **CreateGraphics** der Klasse **Form**. Diese liefert eine neue Instanz der Klasse **Graphics**:

```

public partial class Form1 : Form
{
    ...
    //Eventhandler des Ereignisses Paint
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(Graphics graphics = this.CreateGraphics())
        {
            //Mit dem Graphics-Objekt zeichnen
        }
    }
}

```

Listing B.2: Erzeugung eines Graphics-Objekts mithilfe der Methode CreateGraphics

Wichtig

Die Klasse **Graphics** ist von der Schnittstelle **IDisposable** abgeleitet. Nutzen Sie bei Objekten dieses Typs entweder **using**-Blöcke oder geben Sie das Objekt explizit mit der Methode **Dispose** wieder frei.

B.2.2 Transformation von Objekten

Translation/Verschiebung

Um ein gezeichnetes Objekt zu verschieben, können Sie die Methode **TranslateTransform** der Klasse **Graphics** verwenden:

```

public void TranslateTransform(float x, float y)

```

Dabei wird nicht das Objekt selbst, sondern der Nullpunkt des Koordinatensystems verschoben. Das folgende Beispiel verschiebt den Nullpunkt um 30 Pixel in x-Richtung und 40 Pixel in y-Richtung:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = e.Graphics)
        {
            //Linie ohne Verschiebung
            graphics.DrawLine(new Pen(Color.Black, 1.5F), 50,50,100,100);
            //Verschieben des Nullpunkts
            graphics.TranslateTransform(30,40);
            graphics.DrawLine(new Pen(Color.Red, 1.5F), 50,50,100,100);
        }
    }
}
```

Listing B.3: Verschieben des Nullpunkts

Das Ergebnis sieht folgendermaßen aus:

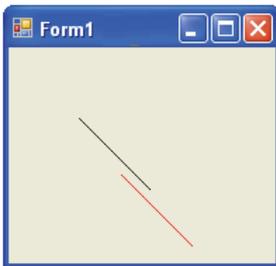


Abb. B.2: Translation der Grafik

Skalierung

Um ein gezeichnetes Objekt zu skalieren, können Sie die Methode **ScaleTransform** der Klasse **Graphics** verwenden:

```
public void ScaleTransform(float scaleX, float scaleY)
```

Das folgende Beispiel führt eine Skalierung um den Faktor 2 in x- und y-Richtung durch:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = e.Graphics)
        {
            //Skalierung um Faktor 2
            graphics.ScaleTransform(2,2);
            graphics.DrawEllipse(new Pen(Color.Red, 1.5F), 0,0,100,100);
        }
    }
}
```

Listing B.4: Skalierung eines Kreises um den Faktor 2

Das Ergebnis sieht entsprechend so aus:

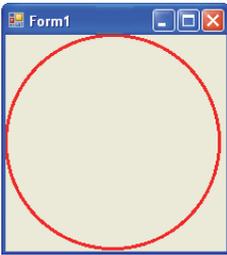


Abb. B.3: Skalierung eines Kreises

Rotation

Um schlussendlich eine Rotation durchzuführen, müssen Sie die Methode **RotateTransform** der Klasse **Graphics** verwenden:

```
public void RotateTransform(float angle)
```

Dabei wird nicht das Objekt, sondern das Koordinatensystem gedreht. Das folgende Beispiel dreht die gezeichnete Linie um 45°:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = e.Graphics)
        {
            //Rotation um 45 Grad und anschließende Translation
            graphics.RotateTransform(45);
            graphics.TranslateTransform(100,0);
            graphics.DrawLine(new Pen(Color.Red, 1.5F), 60,0,100,100);
        }
    }
}
```

Listing B.5: Rotation einer Linie

Das Ergebnis der Rotation sieht folgendermaßen aus:

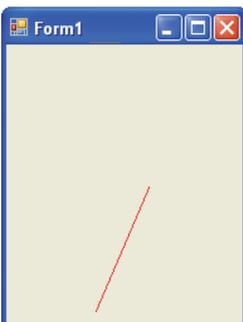


Abb. B.4: Rotation einer Linie

B.3 Mit GDI+ zeichnen

Zum Zeichnen von Linien, Kreisen, Kurven und anderen Objekten müssen Sie die entsprechende Methode der Klasse **Graphics** verwenden. Im Folgenden werden Sie die einzelnen Formen und Methoden näher kennenlernen.

Hinweis

Wir werden hier bereits vorgreifen und die Klassen **Pen** und **Brush** zum Zeichnen verwenden. Diese werden in Abschnitt B.7 genauer betrachtet. Wichtig ist für Sie an dieser Stelle nur, dass damit ein Zeichengerät wie beim Zeichnen auf dem Papier mit dem Bleistift zur Verfügung gestellt wird.

B.3.1 Linien

Um eine Linie zu zeichnen, müssen Sie die Methode **DrawLine** der Klasse **Graphics** verwenden. Diese hat die folgende Definition:

```
public void DrawLine(Pen pen, Point start, Point end)
```

Das folgende Beispiel zeichnet eine rote Linie vom Startpunkt mit den Koordinaten (5,10) zum Endpunkt mit den Koordinaten (50,30):

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        this.Paint += FormPaint;
    }

    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(Graphics graphics = this.CreateGraphics())
        {
            Point startPoint = new Point(5,10);
            Point endPoint = new Point(50,30);
            graphics.DrawLine(new Pen(Color.Red), startPoint, endPoint);
        }
    }
}
```

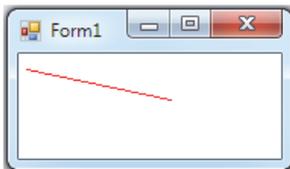
Listing B.6: Zeichnen einer Linie

Wenn Sie den entsprechenden Ausschnitt vergrößern, werden Sie einen unangenehmen Effekt bemerken, bei dem sich einzelne Stufen bilden. Dieser Effekt tritt bei Linien auf, deren Winkel zwischen 0° und 90° beträgt. Das kommt daher, dass nur eine begrenzte Auflösung möglich ist, bei denen die Pixel in das xy-Raster passen müssen. Bei der Neigung kommt es so zu Abweichungen dieses Rasterprinzips und dadurch zu Sprüngen, die den Treppen- bzw. Stufeneffekt hervorrufen. Eine Lösung für dieses Problem stellt die **Anti-aliasing**-Technik dar. Mithilfe dieser Technik werden die Kanten geglättet, indem der Rand der Linie mit Pixeln aufgefüllt wird, deren Farbe sich aus der Linie und der Hintergrundfarbe zusammensetzt.

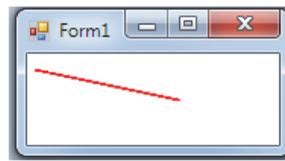
Für den Einsatz dieser Technik stellt Ihnen GDI+ die Eigenschaft **SmoothingMode** zur Verfügung, die die Werte **HighSpeed**, **HighQuality**, **Default**, **Invalid**, **None** und **AntiAlias** erwartet. Dabei stellt **SmoothingMode.HighSpeed** eine schnelle Berechnung dar, deren Ergebnis jedoch kantig ist. Der Wert **SmoothingMode.HighQuality** ist dagegen langsamer in der Berechnung, liefert aber ein saubereres Ergebnis. Mithilfe des Wertes **SmoothingMode.AntiAlias** wird das Antialiasing aktiviert:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = this.CreateGraphics())
    {
        Point startPoint = new Point(5,10);
        Point endPoint = new Point(50,30);
        graphics.SmoothingMode = SmoothingMode.AntiAlias;
        graphics.DrawLine(new Pen(Color.Red), startPoint, endPoint);
    }
}
```

Listing B.7: Verwendung der Antialiasing-Technik beim Zeichnen von Linien



Ohne Antialiasing



Mit Antialiasing

Abb. B.5: Ergebnis mit und ohne Antialiasing

B.3.2 Polyline

Sollen mehrere miteinander verbundene Linien (**Polyline**) gezeichnet werden, müssen Sie die Methode **DrawLines** verwenden. Sie hat folgende Definition:

```
public void DrawLines(Pen pen, Point[] points)
```

Alternativ können Sie auch ein Array von Objekten des Typs **PointF** übergeben. Das folgende Beispiel zeichnet ein Objekt, das sich aus Linien zusammensetzt. Es definiert die Punkte (5,10), (100,30) und (50,50):

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = this.CreateGraphics())
    {
        Point[] points = new Point[]
        {
            new Point(5,10),
            new Point(100,30),
            new Point(50,50)
        }
    }
}
```

```
};  
  
graphics.DrawLine(new Pen(Color.Red), points);  
}  
}
```

Listing B.8: Zeichnen mehrerer Linien

Das Ergebnis sieht folgendermaßen aus:

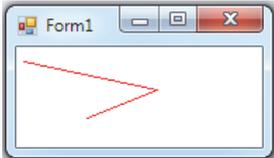


Abb. B.6: Ein Polyline

B.3.3 Polygone

Wollen Sie ein Vieleck zeichnen, das aus n-Ecken besteht, so müssten Sie entsprechend der bisherigen Vorgehensweise mehrere Punkte definieren und diese über `DrawLines` zeichnen lassen. Eine andere Möglichkeit bietet die Klasse `Graphics` über die Methode `DrawPolygon`, die folgende Definition aufweist:

```
public void DrawPolygon(Pen pen, Point[] points)
```

Dabei ist zu beachten, dass das Polygon automatisch ab dem letzten Punkt geschlossen wird. Das folgende Beispiel erzeugt einen Stern:

```
public partial class Form1 : Form  
{  
    ...  
    private void FormPaint(object sender, PaintEventArgs e)  
    {  
        using(Graphics graphics = this.CreateGraphics())  
        {  
            var points = new Point[]  
            {  
                new Point(20,100),  
                new Point(80,100),  
                new Point(100,50),  
                new Point(120,100),  
                new Point(180,100),  
                new Point(120,140),  
                new Point(160,200),  
                new Point(95,170),  
                new Point(30,200),  
                new Point(70,140)  
            };  
            //Zeichnen des Sterns  
            graphics.DrawPolygon(new Pen(Color.Black), points);  
        }  
    }  
}
```

Listing B.9: Zeichnen eines Polygons, hier als Stern

Das Ergebnis sieht nun folgendermaßen aus:

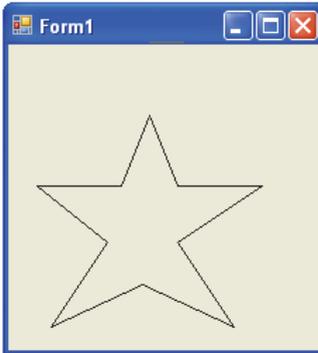


Abb. B.7: Zeichnen eines Sterns mit DrawPolygon

Wenn Sie den Stern jetzt noch entsprechend einfärben möchten, können Sie die Methode **FillPolygon** verwenden, die folgendermaßen definiert ist:

```
public void FillPolygon(Brush brush, Point[] points)
```

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(Graphics graphics = this.CreateGraphics())
        {
            var points = new Point[]
            {
                new Point(20,100),
                new Point(80,100),
                new Point(100,50),
                new Point(120,100),
                new Point(180,100),
                new Point(120,140),
                new Point(160,200),
                new Point(95,170),
                new Point(30,200),
                new Point(70,140)
            };
            //Zeichnen des Sterns mit Füllfarbe
            graphics.FillPolygon(Brushes.Yellow, points);
        }
    }
}
```

Listing B.10: Füllen eines Polygons mit Farbe

Das Ergebnis dieser Anweisung sieht folgendermaßen aus:



Abb. B.8: Mit Farbe ausgefüllter Stern mithilfe der Methode `FillPolygon`

B.3.4 Rechtecke

Eine ähnliche Definition der Methoden wie zum Zeichnen von Polygonen existiert auch für das Zeichnen von Rechtecken. Ein einzelnes Rechteck kann mithilfe der Methode `DrawRectangle` gezeichnet werden. Sie hat folgende Definition:

```
public void DrawRectangle(Pen pen, Rectangle rectangle)
```

Wollen Sie das Rechteck farbig ausfüllen, können Sie die Methode `FillRectangle` verwenden:

```
public void FillRectangle(Brush brush, Rectangle rectangle)
```

Das folgende Beispiel zeigt die Verwendung der Methoden `DrawRectangle` und `FillRectangle`. Dabei wird um das gefüllte Rechteck entsprechend ein Rahmen gezeichnet:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = e.Graphics)
        {
            //Eigenschaften des Rechtecks vorbereiten
            Point rectangleLocation = new Point(50,50);
            Size rectangleSize = new Size(80,50);
            Rectangle rectangle = new Rectangle(rectangleLocation,
            rectangleSize);

            //Rahmen des Rechtecks zeichnen
            graphics.DrawRectangle(new Pen(Color.Black), rectangle);
            //Füllen des Rechtecks
            graphics.FillRectangle(Brushes.Red, rectangle);
        }
    }
}
```

Listing B.11: Zeichnen und Füllen eines Rechtecks

Das Ergebnis sieht folgendermaßen aus:

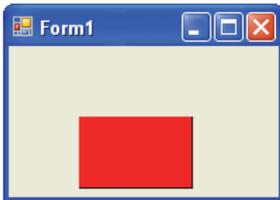


Abb. B.9: Ein rotes Rechteck mit schwarzem Rahmen

Wollen Sie mehrere Rechtecke gleichzeitig zeichnen, sollten Sie die Methoden **DrawRectangles** bzw. **FillRectangles** verwenden:

```
public void DrawRectangles(Pen pen, Rectangle[] rectangles)
public void FillRectangles(Brush brush, Rectangle[] rectangles)
```

Wie Sie sehen, erwarten beide Methoden ein Array aus Rechtecken. Die Verwendung entspricht dabei weitestgehend den Methoden **DrawRectangle** und **FillRectangle**. Deshalb wird hier auf ein entsprechendes Beispiel verzichtet.

B.3.5 Splines

Angenommen, Sie möchten eine Kurve zeichnen, um beispielsweise ein Diagramm darzustellen. Das Problem dabei ist jedoch, dass die Kurve sehr eckig wird, da nur wenige Punkte angegeben werden. Dies bedeutet, dass, wenn Sie die Kurve runder zeichnen wollen, Sie jeden einzelnen Zwischenpunkt angeben müssten. Aber das ist mühselig. Eine andere Möglichkeit bietet sich jedoch durch die Verwendung von Splines. Dabei wird zwischen den einzelnen Punkten, bei denen ein Wechsel des Anstiegs zu verzeichnen ist, ein weicher Übergang berechnet und gezeichnet.

Um ein Spline zu zeichnen, können Sie die Methode **DrawCurve** verwenden:

```
public void DrawCurve(Pen pen, Point[] points)
```

Diese Methode ist überladen und ermöglicht zusätzlich die Angabe ...

- ... des Startpunkts in Form eines Index für das `Point`-Array,
- ... der Anzahl der zu zeichnenden Punkte,
- ... der Spannung, das heißt, wie der Übergang zwischen zwei Teilen geschaffen werden soll.

Das folgende Beispiel zeichnet ein entsprechendes Spline. Dabei soll zunächst dieselbe Kurve mittels der Methode `DrawLines` gezeichnet werden, um Ihnen eine Vergleichsmöglichkeit anbieten zu können:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
```

```

using(Graphics graphics = this.CreateGraphics())
{
    //Punkte für die Kurve definieren
    var points = new Point[]
    {
        new Point(50,100),
        new Point(110,60),
        new Point(170,100)
    };

    //Zeichnen der Kurve mittels DrawLines
    graphics.DrawLines(new Pen(Color.Black, 2.0f), points);

    //Zeichnen der Kurve mittels DrawCurve
    graphics.DrawCurve(new Pen(Color.Red, 2.0f), points);
}
}
}

```

Listing B.12: Zeichnen eines Splines

Das Ergebnis sieht folgendermaßen aus:

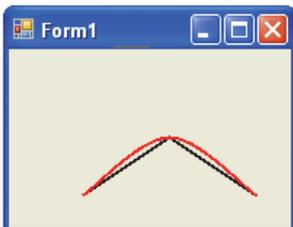


Abb. B.10: Verwendung eines Splines

Wenn Sie die Kurve geschlossen zeichnen wollen, können Sie alternativ auch die Methode **DrawClosedCurve** verwenden:

```
public void DrawClosedCurve(Pen pen, Points[] point)
```

Fügen Sie den Aufruf der Methode entsprechend in den Code aus Listing B.12 und kommentieren Sie die Methoden `DrawLines` und `DrawCurve` aus, dann erhalten Sie folgendes Ergebnis:

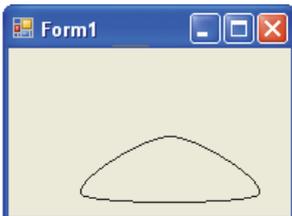


Abb. B.11: Ergebnis der Verwendung der Methode `DrawClosedCurve`

B.3.6 Bézierkurven

Wollen Sie noch feinere Kurven erzeugen, um komplexere Figuren zu zeichnen, empfiehlt sich die Verwendung der Methode **DrawBezier**:

```
public void DrawBezier(Pen pen, Point pt1, Point pt2,  
                      Point pt3, Point pt4)
```

Dabei stellen die Punkte **pt1** den Startpunkt und **pt4** den Endpunkt der Kurve dar. Die Punkte **pt2** und **pt3** dienen als Stützpunkte, mit denen die Linie aufgezogen wird.

Das folgende Beispiel demonstriert die Verwendung der Methode **DrawBezier**:

```
Public partial class Form1 : Form  
{  
    ...  
    private void FormPaint(object sender, PaintEventArgs e)  
    {  
        using(var graphics = e.Graphics)  
        {  
            Point startPoint = new Point(30, 30);  
            Point endPoint = new Point(160, 30);  
            Point firstSupportPoint = new Point(80,60);  
            Point secondSupportPoint = new Point(120,10);  
  
            //Zeichnen der Bezierkurve  
            graphics.DrawBezier(new Pen(Color.Black, 2.0F),  
                                startPoint, firstSupportPoint, secondSupportPoint,  
                                endPoint);  
        }  
    }  
}
```

Listing B.13: Zeichnen einer Bézierkurve

Das Ergebnis der Anweisung sieht folgendermaßen aus:

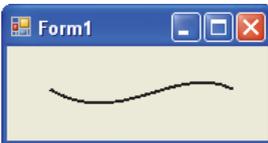


Abb. B.12: Zeichnen einer Bézierkurve

Um mehrere Bézierkurven gleichzeitig zu zeichnen, sollten Sie die Methode **DrawBeziers** verwenden:

```
public void DrawBeziers(Pen pen, Point[] points)
```

Da das **Point**-Array im Vergleich zur Verwendung der Methode **DrawBezier** den einzigen Unterschied darstellt, soll hier auf ein weiteres Beispiel verzichtet werden.

B.3.7 Ellipsen und Kreise

Um eine Ellipse bzw. einen Kreis zu zeichnen, müssen Sie die Methode **DrawEllipse** verwenden:

```
public void DrawEllipse(Pen pen, Rectangle region)
```

Dieser Methode müssen Sie ein Rechteck übergeben, das den Zeichenbereich der Ellipse bzw. des Kreises darstellt. Entsprechend können Sie Ellipsen und Kreise auch farbig ausfüllen. Hierzu existiert die Methode **FillEllipse**:

```
public void FillEllipse(Brush brush, Rectangle region)
```

Das folgende Beispiel zeichnet einen blauen Kreis:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(Graphics graphics = this.CreateGraphics())
        {
            var regionLocation = new Point(50,50);
            var regionSize = new Size(100,100);
            var regionRectangle = new Rectangle(regionLocation,
            regionSize);

            //Zeichnen des Kreisrahmens
            graphics.DrawEllipse(new Pen(Color.Black), regionRectangle);

            //Füllen des Kreises
            graphics.FillEllipse(Brushes.Blue, regionRectangle);
        }
    }
}
```

Listing B.14: Zeichnen eines blauen Kreises

Das Ergebnis sieht folgendermaßen aus:

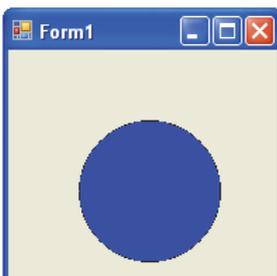


Abb. B.13: Ergebnis für das Zeichnen eines blauen Kreises

B.3.8 Segmente

Wenn Sie je geplant haben, eine Statistik visuell darzustellen, dann haben Sie auch schon an Kuchendiagramme gedacht. Mit den Ihnen bisher bekannten Mitteln ist es jedoch relativ aufwendig, ein entsprechendes Kuchenstück bzw. Segment zu erzeugen. Einfacher ist es dagegen mit der Methode **DrawPie**:

```
public void DrawPie(Pen pen, Rectangle region, float startAngle,
                   float sweepAngle)
```

Mittels dieser Methode wird ein entsprechendes Segment gezeichnet, das durch eine Ellipse und zwei Linien begrenzt ist. Dabei werden diese durch die über **startAngle** (Startwinkel) und **sweepAngle** (Bogenwinkel) gesetzten Winkel berechnet. Die folgende Abbildung soll das Prinzip verdeutlichen:

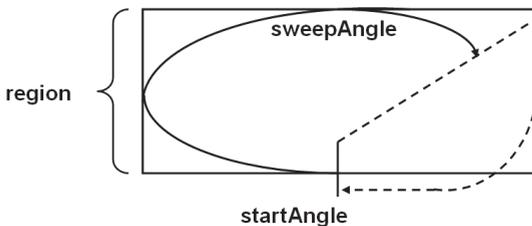


Abb. B.14: Definition eines Segments

Zusätzlich existiert die Methode **FillPie**, mit der Sie ein entsprechendes Segment farbig füllen können:

```
public void FillPie(Brush brush, Rectangle region, float startAngle,
                   float sweepAngle)
```

Das folgende Beispiel erzeugt ein Kuchendiagramm, bestehend aus drei Segmenten:

```
Public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = this.CreateGraphics())
        {
            Point regionLocation = new Point(30,30);
            Size regionSize = new Size(100,100);
            Rectangle regionRectangle = new Rectangle(
                regionLocation, regionSize);

            //Erstes Segment
            graphics.DrawPie(new Pen(Color.Black, 1.5F), regionRectangle,
                10, 80);
            graphics.FillPie(Brushes.Red, regionRectangle, 10, 80);

            //Zweites Segment
```

```
graphics.DrawPie(new Pen(Color.Black, 1.5F), regionRectangle,
90, 40);
graphics.FillPie(Brushes.Green, regionRectangle, 90, 40);

//Drittes Segment
graphics.DrawPie(new Pen(Color.Black, 1.5F), regionRectangle,
130, 240);
graphics.FillPie(Brushes.Yellow, regionRectangle, 130, 240);
}
}
}
```

Listing B.15: Zeichnen eines Kuchendiagramms

Das Ergebnis der Anweisungen sieht folgendermaßen aus:

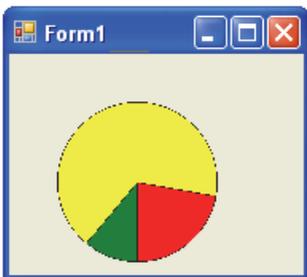


Abb. B.15: Ergebnis des Zeichnens eines Kuchendiagramms

Hinweis

Positive Winkel werden immer im Uhrzeigersinn angegeben.

B.3.9 Bogen

Wenn Sie einen Bogen zeichnen wollen, können Sie die Methode **DrawArc** verwenden:

```
public void DrawArc(Pen pen, Rectangle region, float startAngle,
float sweepAngle)
```

Die Definition der Methode dürfte Ihnen von der Methode **DrawPie** bekannt vorkommen. Im Gegensatz zur Methode **DrawPie** zeichnet die Methode **DrawArc** jedoch lediglich den Bogen, nicht aber die Verbindungen zum Mittelpunkt der Ellipse. Ein Bogen kann entsprechend nicht gefüllt werden.

Das folgende Beispiel zeichnet einen Bogen:

```
public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = this.CreateGraphics())
```

```

{
    Point regionLocation = new Point(30,30);
    Size regionSize = new Size(100,100);
    var region = new Rectangle(regionLocation, regionSize);

    //Zeichnen des Bogens
    graphics.DrawArc(new Pen(Color.Red, 2.0F), region,
    90, 140);
}
}
}

```

Listing B.16: Zeichnen eines Bogens

Das Ergebnis zum Zeichnen des Bogens sieht so aus:

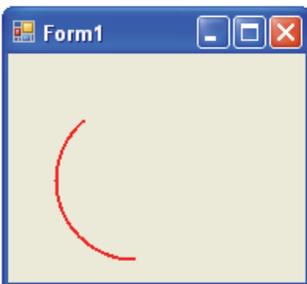


Abb. B.16: Zeichnen eines Bogens

B.3.10 Pfad

Komplexere Figuren setzen sich aus verschiedenen Linien und Kurven zusammen. Dabei reicht es grundsätzlich, ein Skelett bzw. einen Umriss zu zeichnen. Dieser Umriss stellt einen Pfad dar. Um einen entsprechenden Pfad zu zeichnen, können Sie die Methode **DrawPath** verwenden:

```
public void DrawPath(Pen pen, GraphicsPath path)
```

Wie Sie sehen, erwartet die Methode eine Instanz der Klasse **GraphicsPath**. Diese definiert den Pfad und stellt selbst Methoden wie beispielsweise `AddLine` oder `AddRectangle` zur Verfügung, um aus verschiedenen Formen einen Pfad und damit ein komplexes Objekt zu erzeugen. Im folgenden Beispiel wird ein Männchen erzeugt und gezeichnet:

```

public partial class Form1 : Form
{
    ...
    private void FormPaint(object sender, PaintEventArgs e)
    {
        using(var graphics = e.Graphics)
        {
            GraphicsPath path = new GraphicsPath();
            //1.) Zeichnen des Kopfes
            var headRegion = new Rectangle(

```

```
new Point(100,30), new Size(50,50));
path.AddEllipse(headRegion);

//2.) Zeichnen der Augen
//Da ein neuer Teil separate gezeichnet werden soll,
//muss zunächst die Methode StartFigure aufgerufen
//werden
path.StartFigure();
var leftEyeRegion = new Rectangle(
new Point(110,50), new Size(5,5));
var rightEyeRegion = new Rectangle(
new Point(130, 50), new Size(5,5));
path.AddEllipse(leftEyeRegion);
path.AddEllipse(rightEyeRegion);
graphics.FillEllipse(Brushes.Black, leftEyeRegion);
graphics.FillEllipse(Brushes.Black, rightEyeRegion);

//3.) Zeichnen der Nase
path.StartFigure();
path.AddLine(new Point(122,60), new Point(122,68));

//4.) Zeichnen des Mundes
path.StartFigure();
var mouthRegion = new Rectangle(
new Point(115,65), new Size(20,10));
path.AddArc(mouthRegion, 360, 180);

//5.) Zeichnen des Körpers
path.StartFigure();
Point[] bodyPoints = new Point[]
{
    new Point(110,75),
    new Point(40,100),
    new Point(60,120),
    new Point(100,100),
    new Point(90,180),
    new Point(160,180),
    new Point(150,100),
    new Point(190,120),
    new Point(210,100),
    new Point(140,75)
};
path.AddLines(bodyPoints);

//6.) Zeichnen der Beine
path.StartFigure();
var footPoints = new Point[]
{
    new Point(90,180),
    new Point(70,240),
    new Point(110,240),
    new Point(125,200),
```

```
        new Point(140,240),
        new Point(180,240),
        new Point(160,180)
    };
    path.AddLines(footPoints);

    //7.) Zeichnen der linken Hand
    path.StartFigure();
    var leftHandRegion = new Rectangle(
    new Point(35, 105), new Size(20,20));
    path.AddArc(leftHandRegion, 0, 270);

    //8.) Zeichnen der rechten Hand
    path.StartFigure();
    var rightHandRegion = new Rectangle(
    new Point(195, 105), new Size(20,20));
    path.AddArc(rightHandRegion, 270, 270);

    //9.) Zeichnen des linken Schuhs
    path.StartFigure();
    var leftShoeRegion = new Rectangle(
    new Point(60, 240), new Size(20,20));
    path.AddArc(leftShoeRegion, 80, 200);
    path.StartFigure();
    var leftShoePoints = new Point[]
    {
        new Point(70,260),
        new Point(100,260),
        new Point(110,240),
    };
    path.AddLines(leftShoePoints);

    //10.) Zeichnen des rechten Schuhs
    path.StartFigure();
    var rightShoeRegion = new Rectangle(
    new Point(170, 240), new Size(20,20));
    path.AddArc(rightShoeRegion, 260, 200);
    path.StartFigure();
    var rightShoePoints = new Point[]
    {
        new Point(180,260),
        new Point(150,260),
        new Point(140,240),
    };
    path.AddLines(rightShoePoints);

    //11.) Zeichnen des Pfades
    graphics.DrawPath(new Pen(Color.Black, 2.0F), path);
    }
    }
}
```

Listing B.17: Zeichnen eines Männchens mithilfe der Methode DrawPath

Das entsprechende Ergebnis sieht so aus:

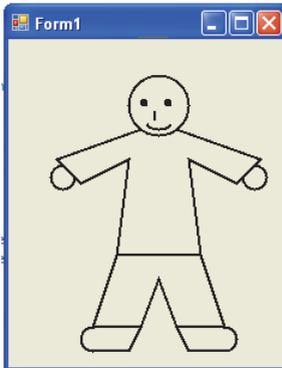


Abb. B.17: Zeichnen eines Männchens

Hinweis

Wollen Sie den ganzen Pfad einfärben, können Sie die Methode `FillPath` verwenden.

B.4 Mit Bildern arbeiten

Eine weitere Möglichkeit für die Grafikprogrammierung stellt die Bildbearbeitung bzw. Bildmanipulation dar. Die hierfür notwendigen Klassen befinden sich im Namensraum `System.Drawing.Imaging`.

Um mit Bildern zu arbeiten, benötigen Sie ein Objekt der Klasse `Image`. Der Abschnitt zeigt Ihnen, wie Sie mit Bildern arbeiten können.

B.4.1 Das Steuerelement PictureBox

Das Steuerelement `PictureBox` sollte Ihnen noch aus dem Kapitel zu Windows Forms bekannt sein. Damit können Sie Bilder auf einem Formular anzeigen. Es definiert die Eigenschaft `Image`, der ein Objekt des Typs `Image` zugewiesen werden kann. Ab jetzt wird zur Ausgabe von Grafiken stets das Steuerelement `PictureBox` verwendet. Betrachten Sie aber zunächst einmal die eigentliche Klasse `Image`.

B.4.2 Die Klasse Image

Sie können vorerst kein Objekt der Klasse `Image` erzeugen, da der Konstruktor der Klasse als `internal` markiert ist. Dennoch gibt es zwei Möglichkeiten, um an ein solches Objekt zu gelangen: Entweder es existiert bereits ein fertiges Bild innerhalb der `PictureBox` und Sie können sich dieses mittels der Eigenschaft `Image` beschaffen, oder Sie erzeugen ein neues Objekt durch die Instanziierung und Zuweisung der Klasse `Bitmap`:

```
Image image = new Bitmap(50,50);
```

Haben Sie ein solches Objekt erzeugt, müssen Sie es natürlich für die Anzeige wieder der Eigenschaft `Image` der `PictureBox` zuweisen. Im folgenden Abschnitt werden Sie die wichtigsten Eigenschaften und Methoden der Klasse **Image** kennenlernen.

B.4.3 Eigenschaften und Methoden der Klasse Image

Die folgenden Tabellen bieten eine Übersicht über die wichtigsten Eigenschaften und Methoden der Klasse **Image**. Dabei wird davon ausgegangen, dass bereits eine `PictureBox` mit der Bezeichnung `pctBox` existiert:

Ermittlung der Bildgröße und der Auflösung

Eigenschaft/Methode	Beispiel	Bedeutung
<code>int Height</code>	<pre>int currentPictureHeight = pctBx.Image.Height; //Ausgabe für ein Bild der //Größe 50 x 50: 50 MessageBox.Show (currentPictureHeight. ToString());</pre>	Liefert die Höhe des Bildes in Pixel zurück.
<code>int Width</code>	<pre>int currentPictureWidth = pctBx.Image.Width; //Ausgabe für ein Bild der //Größe 50 x 50: 50 MessageBox.Show (currentPictureWidth. ToString());</pre>	Liefert die Breite des Bildes in Pixel zurück.
<code>Size Size</code>	<pre>Size imageSize = pctBx.Image.Size; //Ausgabe für ein Bild der //Größe 100 x 100: 100, 100 MessageBox.Show(imageSize.Width + ", " + imageSize.Height);</pre>	Liefert die Breite und Höhe des Bildes in Pixel zurück.
<code>float HorizontalResolution</code>	<pre>float horizontalResolution = pctBx.Image. HorizontalResolution; //Ausgabe: 96 MessageBox.Show (horizontalResolution. ToString());</pre>	Liefert die horizontale Auflösung des Bildes in Pixel pro Zoll zurück.
<code>float VerticalResolution</code>	<pre>float verticalResolution = pctBx.Image. VerticalResolution; //Ausgabe: 96 MessageBox.Show (verticalResolution. ToString());</pre>	Liefert die vertikale Auflösung des Bildes in Pixel pro Zoll zurück.

Tabelle B.1: Die wichtigsten Eigenschaften und Methoden zur Ermittlung der Bildgröße und der Auflösung eines Bildes

Eigenschaft/Methode	Beispiel	Bedeutung
static int GetPixelFormatSize (PixelFormat pixfmt)	int formatSize = Image.GetPixelFormatSize(PixelFormat.Alpha); //Ausgabe: false MessageBox.Show(formatSize);	Liefert die Farbtiefe für das angegebene Format als Anzahl der Bits pro Pixel zurück.

Tabelle B.1: Die wichtigsten Eigenschaften und Methoden zur Ermittlung der Bildgröße und der Auflösung eines Bildes (Forts.)

Ermittlung des Bildformats

Eigenschaft	Beispiel	Bedeutung
PixelFormat PixelFormat	//Ausgabe: Format24bppRgb MessageBox.Show(pctBx.Image.PixelFormat.ToString());	Liefert das Pixelformat des Bildes zurück.
ImageFormat RawFormat	//Ausgabe: [ImageFormat: //b96b3cae-0728-11d3- //9d7b-0000f81ef32e] MessageBox.Show(pctBx.Image.RawFormat.ToString());	Liefert das Dateiformat des Bildes zurück. Die Enumeration ImageFormat definiert verschiedene Bildformate, von denen folgende die wichtigsten darstellen: <ul style="list-style-type: none"> ■ Bmp: Bitmap-Format ■ Gif: GIF-Format ■ Icon: Windows-Symbol ■ Jpeg: JPEG-Format ■ Png: PNG-Format ■ Tiff: TIFF-Format

Tabelle B.2: Die wichtigsten Eigenschaften und Methoden zur Ermittlung des Bildformats

Eigenschaft	Beispiel	Bedeutung
<code>static bool IsAlphaPixelFormat(PixelFormat pixfmt)</code>	<code>//Ausgabe: true MessageBox.Show (Image.IsAlphaPixelFormat (PixelFormat.Alpha);</code>	Gibt einen Wahrheitswert (true/false) zurück, der aussagt, ob das angegebene Pixelformat Alphainformationen enthält.

Tabelle B.2: Die wichtigsten Eigenschaften und Methoden zur Ermittlung des Bildformats

Bilder laden, drehen und speichern

Methode	Beispiel	Bedeutung
<code>static Image FromFile(string fileName)</code>	<code>//Laden des Testbildes pctBx.Image = Image. FromFile("C:\\Test.bmp");</code>	Erzeugt ein Image für die angegebene Datei.
<code>static Image FromStream(Stream stream)</code>	<code>//Erzeugen eines FileStreams FileStream stream = new File- Stream("C:\\Test.bmp", FileStream.Open, FileAccess. Read); pctBx.Image = Image.From- Stream(stream);</code>	Erzeugt ein Image aus dem gegebenen Stream.
<code>void RotateFlip(RotateFlipType rotateFlipType)</code>	<code>//Drehen des Bildes um 90 //Grad im Uhrzeigersinn ohne //Kippen pctBx.Image. RotateFlip(RotateFlipType. Rotate90FlipNone);</code>	Dreht bzw. kippt das Bild.
<code>void Save(string fileName [,ImageFormat format])</code>	<code>//Speichern eines Bitmap- //Bildes als Jpeg-Bild pctBx.Image. Save("C:\\Converted.jpg", ImageFormat.Jpeg);</code>	Speichert das Bild unter dem angegebenen Pfad ab. Optional können Sie auch ein anderes Bildformat angeben. Dadurch können Sie beispielsweise ein Bild im Format BMP in das Format JPEG umwandeln.

Tabelle B.3: Die wichtigsten Methoden zum Laden, Speichern und Drehen von Bildern

Vorsicht

Die Maße eines **Image**-Objekts entsprechen in keinem Fall den Maßen der **PictureBox**.

Hinweis

Die Klasse **Graphics** definiert die Methode **FromImage**, der Sie ein **Image** übergeben müssen. Der Rückgabewert ist ein **Graphics**-Objekt, mit dem Sie auf das übergebene Bild zeichnen können.

B.4.4 Transformation von Bildern

Wie ein Bild gedreht wird, haben Sie bereits anhand der Methode **RotateFlip** gesehen. Wenn Sie ein vorhandenes Bild skalieren möchten, müssen Sie dazu ein neues Objekt der Klasse **Bitmap** erzeugen und diesem das Originalbild und die neuen Bildmaße übergeben. Danach weisen Sie das neue Objekt der Eigenschaft **Image** der **PictureBox** zu:

```
...
//Vergrößert das Bild um den angegebenen Faktor
public void ScaleImage(int scaleFactor)
{
    Size scaledSize = new Size(pctBx.Image.Width * scaleFactor,
    pctBx.Image.Height * scaleFactor);
    pctBx.Image = new Bitmap(pctBx.Image, scaledSize);
}
}
```

Listing B.18: Skalieren eines Bildes

B.4.5 Punkte ermitteln und setzen

Mithilfe der Methode **Color GetPixel(int x, int y)** erhalten Sie den Farbwert des Pixels an der durch **x** und **y** angegebenen Position innerhalb des Koordinatensystems. Umgekehrt können Sie mithilfe der Methode **void SetPixel(int x, int y, Color color)** den Punkt an der durch **x** und **y** angegebenen Position in der durch **color** angegebenen Farbe setzen. Diese Methoden gehören zur Klasse **Bitmap**. Das folgende Beispiel färbt jeden vierten Punkt eines Bildes Rot, alle anderen Punkte, die den Farbwert weiß haben, werden mit Schwarz gekennzeichnet:

```
...
private void ButtonDraw_Click(object sender, EventArgs e)
{
    int imageWidth = pctBx.Image.Width;
    int imageHeight = pctBx.Image.Height;
    Bitmap bitmap = new Bitmap(pctBx.Image, imageWidth,
    imageHeight);

    for(int i = 0; i < bitmap.Width; i++)
    {
        for(int j = 0; j < bitmap.Height; j++)
            if(i % 4 == 0)
                bitmap.SetPixel(i,j,Color.Red); //Jedes vierte Pixel
            else if(bitmap.GetPixel(i,j).ToArgb() ==
                Color.White.ToArgb())
                {

```

```

        //Weiße Pixel schwarz färben
        bitmap.SetPixel(i,j,Color.Black);
    }
}

pctBx.Image = bitmap;
pctBx.Refresh();
}

```

Listing B.19: Ermitteln und Setzen von Pixeln in einem Bild

B.5 Textausgabe

Um nicht nur einzelne Grafiken, sondern auch Text in Form einer Grafik ausgeben zu können, benötigen Sie die Klassen aus dem Namensraum **System.Drawing.Text**. Zur Ausgabe eines Textes müssen Sie die Methode **DrawString** der Klasse **Graphics** verwenden:

```
public void DrawString(string text, Font font, Brush brush, PointF xy)
```

Zunächst müssen Sie der Methode einen Text übergeben. Anschließend übergeben Sie ein Objekt der Klasse **Font**, die Informationen zur Schrift definiert. Anschließend müssen Sie die Textfarbe in Form eines **Brush**-Objekts angeben. Weiterhin müssen Sie einen Punkt definieren, an dem der Text gezeichnet werden soll. Das folgende Beispiel gibt den Text **Hello World** als Grafik auf dem Formular aus:

```

...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(var graphics = e.Graphics)
    {
        graphics.DrawString("Hello World", new Font("Arial", 12.0F),
            Brushes.Black, new PointF(30.0F, 30.0F));
    }
}

```

Listing B.20: Ausgabe eines Textes als Grafik auf dem Formular

Das Ergebnis der Ausgabe sieht so aus:



Abb. B.18: Ausgabe des Textes als Grafik

Um einen Text beispielsweise zentriert auszugeben, müssen Informationen darüber vorliegen, wie hoch und breit der Text ist. Zur Ermittlung der Höhe und Breite eines Textes können Sie die Methode **MeasureString** verwenden:

```
Public SizeF MeasureString(string text, Font font)
```

Diese Methode liefert die Größe in Form eines Objekts des Typs **SizeF** zurück. Das folgende Beispiel ermittelt die Größe des Textes und zentriert ihn anschließend:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(var graphics = e.Graphics)
    {
        Font textFont = new Font("Arial", 14);
        SizeF currentTextSize = graphics.MeasureString(
            "Hello World", textFont);
        graphics.DrawString("Hello World", textFont,
            Brushes.Black,
            new PointF((this.Width - currentTextSize.Width)/2, 30.0F));
    }
}
```

Listing B.21: Bestimmung der Größe eines Textes und Zentrierung innerhalb des Formulars

B.5.1 Fonts

Die Klasse **Font** definiert die zu verwendende Schrift. Dabei beinhaltet die Klasse eine Vielzahl von verschiedenen Konstruktoren, von denen die folgenden die wichtigsten zur Erzeugung einer Schrift sind:

- **Font(string familyName, float emSize)**
- **Font(string familyName, float emSize, FontStyle style)**

Unter **familyName** müssen Sie die Schriftart angeben. Die Schriftarten sollten Ihnen beispielsweise aus MICROSOFT WORD bekannt sein. Sie werden als String angegeben. Der Parameter **emSize** definiert die Schriftgröße. Zusätzlich können Sie über **style** einen Member der Enumeration **FontStyle** übergeben. Mithilfe dieser Enumeration können Sie den Text entsprechend formatieren. Beispielsweise können Sie ihn fett und unterstrichen darstellen. Im Folgenden werden die Members der Enumeration **FontStyle** gezeigt:

- **Bold**: Fett
- **Italic**: Kursiv
- **Regular**: Normal
- **StrikeOut**: Durchgestrichen
- **Underline**: Unterstrichen

Wollen Sie verschiedene Werte kombinieren, müssen Sie sie durch ein bitweises Oder verknüpfen. Das folgende Beispiel erzeugt einen Text, der die Schrift VERDANA und die Größe 14 hat. Weiterhin wird der Text fett und unterstrichen dargestellt:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(var graphics = e.Graphics)
    {
        Font textFont = new Font("Verdana", 14,
            FontStyle.Bold | FontStyle.Underline);
        graphics.DrawString("Hello World", textFont,
            Brushes.Black, 30.0F, 30.0F);
    }
}
```

Listing B.22: Verwenden einer definierten Schrift zur Darstellung eines Textes

Das Ergebnis sieht entsprechend so aus:

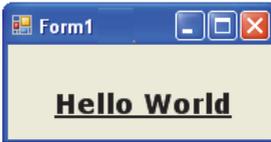


Abb. B.19: Ergebnis des Beispiels zur Definition einer eigenen Schrift

B.6 Farbangaben

Was wäre eine Anwendung bzw. eine Grafik ohne Farben? Auch mit GDI+ ist es möglich, verschiedene Farben zu verwenden. Dabei gibt es verschiedene vom Betriebssystem bereitgestellte Farben. Zur Angabe der vordefinierten Farben können Sie entweder die statische Klasse **SystemColors** oder die Struktur **Color** verwenden. Der folgende Abschnitt soll Sie mit den verschiedenen Farbangaben vertraut machen.

B.6.1 Die Struktur Color

Die Struktur **Color** ist eine statische Struktur und definiert verschiedene Methoden zur Konvertierung von Farben sowie verschiedene Farbkonstanten, wie beispielsweise Rot oder Schwarz, die Sie über **Color.Red** bzw. **Color.Black** verwenden können und deren RGB-Wert (siehe Abschnitt B.6.2) fest definiert ist.

Zu den wichtigsten Methoden zählen:

- **Color.FromArgb(int alpha, int red, int green, int blue)**: Liefert eine Instanz der Struktur **Color** aus den übergebenen ARGB-Farbwerten zurück.
- **Color.FromName(string name)**: Liefert eine Instanz der Struktur **Color** aus dem angegebenen Namen zurück.

Im Folgenden wird das Farbsystem von GDI+ genauer erläutert.

B.6.2 Das ARGB-Farbsystem

Das grundlegende Farbsystem setzt sich aus den drei Grundfarben Rot, Grün und Blau zusammen, die durch additive Überlagerung zu einer Farbe zusammengesetzt werden. Dieses Farbsystem wird **RGB** genannt. Jeder Farbanteil ist dabei in 256 Farbabstufungen unterteilt, was zu einer maximalen Anzahl von $2^{256} = 16$ Mio. Farben führt. Um Transparenz zu ermöglichen, wird ein weiterer Farbkanal hinzugefügt: der **Alpha**-Kanal. Somit entsteht das System **ARGB**, das in GDI+ verwendet wird. Hier wird jede Farbinformation als **ARGB**-Wert (4 Byte) definiert. Die Darstellung eines **ARGB**-Werts erfolgt hexadezimal. Dabei stellen die ersten beiden Hexziffern den Alphawert dar, die anderen, niederwertigen Bytes stellen das **RGB**-System dar.

Die Farbe Pink wird hexadezimal so dargestellt:



Abb. B.20: Das ARGB-System am Beispiel der Farbe Pink

Hinweis

Der hexadezimale Wert `0x00000000` stellt die Farbe Schwarz, der Wert `0x00FFFFFF` die Farbe Weiß, der Wert `0x000000FF` die Farbe Rot, der Wert `0x0000FF00` die Farbe Grün und der Wert `0x00FF0000` die Farbe Blau dar.

Tipp

Wollen Sie ein Graustufenbild erzeugen, müssen die verschiedenen Farbkanäle den gleichen Wert beinhalten.

Vorsicht

Nur Bilder mit dem Format **PNG** ermöglichen eine transparente Darstellung, da nur dieses Format die zusätzliche Information über den Alphawert speichern kann.

B.7 Zeichnen mittels Pinsel und Stift

Wenn Sie ein Bild auf ein Blatt Papier malen wollen, brauchen Sie immer einen Stift oder irgendein anderes Mittel, um Zeichnen zu können. In den bisherigen Beispielen zu GDI+ haben Sie bereits einen Stift (**Pen**) verwendet. Dagegen ist es einfacher, Bilder mittels eines Pinsels (**Brush**) einzufärben, da dieser eine größere Fläche abdeckt, die zu füllen mit einem Stift mühsamer wäre. In diesem Abschnitt des Kapitels werden die »Zeichenwerkzeuge« **Pen** und **Brush** näher betrachtet.

Wichtig

Mit Stiften definieren Sie den Rahmen, mit Pinseln (**Brush**) füllen Sie die Figur. Ein Füllen mithilfe eines Stifts ist nicht möglich.

B.7.1 Die Klasse Pen

Die Klasse **Pen** stellt einen einfachen Stift dar. Um einen Stift zu erzeugen, müssen Sie dem Konstruktor entweder eine Farbe der Struktur `Color` oder eine Farbe der Klasse `Brush` übergeben. Optional erwartet der Konstruktor die Angabe der Liniendicke. Im folgenden Beispiel wird ein roter Stift mit der Dicke 2 Pixel erzeugt:

```
...
var pen = new Pen(Color.Red, 2.0F);
```

Listing B.23: Erzeugung eines Pen-Objekts

Sie können jedoch auch die Methode `FromArgb` der Struktur `Color` verwenden, um einen gemischtfarbigen Stift zu erzeugen. Die folgende Tabelle zeigt die wichtigsten Eigenschaften der Klasse `Pen`:

Eigenschaft	Bedeutung
Brush Brush	Legt die Farbe fest bzw. liefert diese zurück.
Color Color	Legt die Farbe fest bzw. liefert diese zurück.
PenAlignment Alignment	Legt die Ausrichtung des Stifts fest bzw. liefert diese zurück. Die Enumeration PenAlignment definiert dabei die Members: <ul style="list-style-type: none"> ■ Center: Zentriert auf der Linie ■ Inset: Auf der Innenseite der Linie ■ Outset: Außerhalb der Linie ■ Left: Links von der Linie ■ Right: Rechts von der Linie
DashCap DashCap	Legt den Stil für die Enden der gestrichelten Linie fest bzw. liefert diesen zurück. Die Enumeration DashCap definiert die folgenden Members: <ul style="list-style-type: none"> ■ Flat: Die Enden sind quadratisch. ■ Round: Die Enden sind rund. ■ Triangle: Die Enden werden als Dreieck dargestellt.
DashStyle DashStyle	Legt den Stil für die Linie fest bzw. liefert diesen zurück. Die Enumeration DashStyle definiert die folgenden Members: <ul style="list-style-type: none"> ■ Solid: Durchgezogene Linie ■ Dash: Gestrichelte Linie ■ Dot: Aus Punkten bestehende Linie ■ DashDot: Aus Strich-Punkt-Paaren bestehende Linie ■ DashDotDot: Aus Strich-Punkt-Punkt-Paaren bestehende Linie ■ Custom: Benutzerdefinierte Linie
LineCap EndCap	Legt den Stil für die Enden der Linie fest bzw. liefert diesen zurück. Die Enumeration LineCap definiert die folgenden Members: <ul style="list-style-type: none"> ■ Flat: Abgeflachtes Linienende ■ Square: Quadratisches Linienende ■ Round: Rundes Linienende ■ Triangle: Aus Dreiecken bestehendes Linienende ■ NoAnchor: Es wird kein Anker verwendet. ■ SquareAnchor: Quadratisches Ankerlinienende ■ RoundAnchor: Rundes Ankerende ■ DiamondAnchor: Rautenförmiges Ankerende ■ ArrowAnchor: Pfeilförmiges Ankerende ■ Custom: Benutzerdefiniertes Linienende ■ AnchorMask: Gibt eine Maske an, mit der geprüft wird, ob es sich bei dem Ende um einen Anker handelt.

Tabelle B.4: Übersicht der Eigenschaften der Klasse `Pen`

Eigenschaft	Bedeutung
LineJoin LineJoin	Legt die Verbindungsart zweier aufeinanderfolgender Linien fest bzw. liefert diese zurück. Die Enumeration LineJoin definiert die folgenden Members: <ul style="list-style-type: none"> ■ Miter: Erzeugt eine scharfe bzw. abgeschnittene Ecke. ■ Bevel: Erzeugt eine diagonale Ecke. ■ Round: Erzeugt einen runden Bogen. ■ MiterClipped: Erzeugt eine scharfe bzw. abgeschrägte Ecke.
float MiterLimit	Legt die Stärkenbegrenzung für die Verbindung von zwei angeschrägten Ecken fest bzw. liefert diese zurück.
PenType PenType	Liefert den Stil des Stifts zurück. Die Enumeration PenType definiert die folgenden Members: <ul style="list-style-type: none"> ■ SolidColor: Durchgehende Füllung ■ HatchFill: Schraffurfüllung ■ TextureFill: Bitmap-Strukturfüllung ■ PathGradient: Pfadverlaufsfüllung ■ LinearGradient: Füllung mit linearem Farbverlauf
LineCap StartCap	Legt den Stil für den Anfang der Linie fest bzw. liefert diesen zurück.
float Width	Legt die Breite des Stifts fest bzw. liefert diese zurück.

Tabelle B.4: Übersicht der Eigenschaften der Klasse Pen (Forts.)

Definieren Sie nun einen roten Stift der Dicke 10 Pixel, dessen Linienstart ein Dreieck und dessen Ende ein Pfeil darstellt und der in der Mitte einen rechten Winkel definiert:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(var graphics = e.Graphics)
    {
        var pen = new Pen(Color.Red, 10.0F);
        pen.StartCap = LineCap.Triangle; //Dreieck
        pen.EndCap = LineCap.ArrowAnchor; //Pfeil
        pen.LineJoin = LineJoin.Miter;

        var penPoints = new Point[]
        {
            new Point(10,50),
            new Point(60,50),
            new Point(60,100)
        };
        graphics.DrawLine(pen, penPoints);
    }
}
```

Listing B.24: Zeichnen mithilfe eines eigenen Stifts

Das Ergebnis der o.g. Anweisungen sieht folgendermaßen aus:

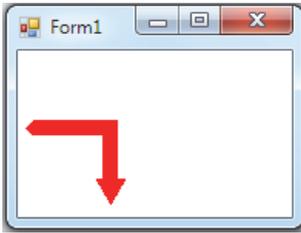


Abb. B.21: Zeichnen mithilfe eines benutzerdefinierten Stifts

B.7.2 Die Klasse Brush

Die Klasse **Brush** stellt einen Pinsel dar, mit dem Figuren und Formen mit Farbe gefüllt werden können. Die Klasse selbst ist abstrakt und dient zur Ableitung verschiedener Pinselarten. Die nachfolgend beschriebenen Pinselarten sind alle von dieser Klasse abgeleitet. Pinsel werden dann eingesetzt, wenn eine `FillXXX`-Methode der Klasse `Graphics` verwendet wird und diese einen entsprechenden Pinsel erwartet. Eine weitere Klasse ist die Klasse **Brushes**, die wie `Color` eine Reihe von Farbkonstanten definiert. Der Aufruf erfolgt dabei statisch.

B.7.3 SolidBrush

Ein **SolidBrush** definiert einen einfarbigen Pinsel. Der Konstruktor erwartet einen Parameter vom Typ `Color`:

```
...
SolidBrush brush = new SolidBrush(Color.Red); //Roter Pinsel
```

B.7.4 LinearGradientBrush

Wenn Sie einen Farbverlauf definieren wollen, müssen Sie die Klasse **LinearGradientBrush** verwenden. Dabei geben Sie zwei Farben und eine Richtung an. Daraus berechnet GDI+ dann den Verlauf:

```
LinearGradientBrush(Rectangle region, Color startColor,
Color endColor, LinearGradientMode linearGradientMode)
```

Das Rechteck **region** stellt den Bereich dar, in dem der Verlauf berechnet wird. Die Farben **startColor** und **endColor** stellen ARGB-Farbwerte dar. Die Enumeration **LinearGradientMode** definiert die Richtung des Verlaufs und beinhaltet die folgenden Members:

- **BackwardDiagonal**: Verlauf von rechts oben nach links unten
- **ForwardDiagonal**: Verlauf von links oben nach rechts unten
- **Horizontal**: Verlauf von links nach rechts
- **Vertical**: Verlauf von oben nach unten

Im folgenden Beispiel wird ein Rechteck mit einem Verlauf von links oben nach rechts unten mit den Farben Blau auf Weiß gezeichnet:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = e.Graphics)
    {
        Rectangle rectangle = new Rectangle(30,30,100,100);
        //Definition des Verlaufs
        var linearGradientBrush = new LinearGradientBrush(
            rectangle, Color.Blue, Color.White,
            LinearGradientMode.ForwardDiagonal);

        graphics.FillRectangle(linearGradientBrush, rectangle);
    }
}
```

Listing B.25: Zeichnen eines Rechtecks mit Verlauf

Das Ergebnis sieht so aus:

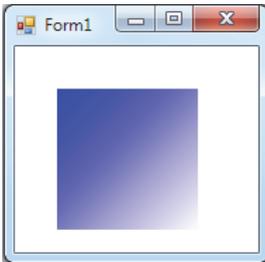


Abb. B.22: Verwendung eines LinearGradientBrush

B.7.5 TextureBrush

In der 3D-Welt werden Objekte normalerweise mit Bildern, sogenannten **Texturen** überzogen. So wird zum Beispiel ein dreidimensionales Rechteck mit einem Bild aus Steinen überzogen, um so eine Mauer darzustellen. Einen ähnlichen Effekt generiert die Klasse **TextureBrush**. Dabei wird eine bestimmte Fläche mit einem Bitmap-Muster überzogen. Ist das Muster zu klein für die Fläche, wird das Bild entsprechend wiederholt.

Im folgenden Beispiel wird ein Rechteck gezeichnet, das ein Bild einer Blume als Textur erhält:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = e.Graphics)
    {
        Rectangle rectangle = new Rectangle(30,30,200,200);
        //Definition der Textur
        var textureBrush = new TextureBrush(
            Image.FromFile("C:\\Flower.bmp"));

        graphics.FillRectangle(textureBrush, rectangle);
    }
}
```

Listing B.26: Zeichnen eines Rechtecks mit einer Textur

Das Ergebnis sieht folgendermaßen aus:

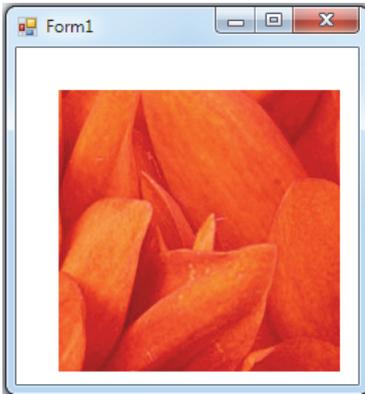


Abb. B.23: Zeichnen eines Rechtecks mit einer Textur

B.7.6 PathGradientBrush

Wollen Sie innerhalb eines Pfades unterschiedliche Farbverläufe definieren, müssen Sie die Klasse **PathGradientBrush** verwenden. Dabei definiert diese Klasse die folgenden Eigenschaften:

- Color **CenterColor**: Die zentrale Farbe, von der gestartet wird
- Color[] **SurroundColors**: Die Farben für die Ecken des Pfades
- PointF **CenterPoint**: Der Mittelpunkt des Pfades

Die Klasse erwartet darüber hinaus im Konstruktor den Pfad. Das folgende Beispiel zeichnet mittels der Klasse **GradientPath** ein Dreieck und verwendet dazu verschiedene Farben, um unterschiedliche Verläufe zu erzeugen:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = e.Graphics)
    {
        //Dreieck definieren
        var path = new GradientPath();
        path.AddLine(50,100,100,50);
        path.AddLine(100,50,150,100);
        path.AddLine(150,100,50,100);

        //Brush definieren
        var pathBrush = new PathGradientBrush(path);
        pathBrush.CenterColor = Color.Yellow;
        pathBrush.SurroundColors = new Color[]{Color.Blue,
        Color.Red, Color.Green};
        pathBrush.CenterPoint = new PointF(100,100);

        //Zeichnen des Dreiecks
        graphics.FillPath(pathBrush, path);
    }
}
```

Listing B.27: Zeichnen unterschiedlicher Farbverläufe entlang eines Pfades

Das Ergebnis sieht so aus:

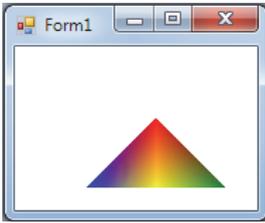


Abb. B.24: Beispiel eines Dreiecks mit PathGradientBrush

B.7.7 HatchBrush

Die letzte Pinselvariation stellt die Klasse **HatchBrush** dar. Ein **HatchBrush** setzt sich aus einem Vordergrundmuster mit eigener Farbe und einer Hintergrundfarbe zusammen und stammt aus dem Namensraum **System.Drawing.Drawing2D**:

HatchBrush(HatchStyle hatchStyle, Color foreColor, Color backColor)

Das folgende Beispiel erzeugt ein Rechteck, das mit einem **HatchBrush** gefüllt ist:

```
...
private void FormPaint(object sender, PaintEventArgs e)
{
    using(Graphics graphics = e.Graphics)
    {
        //Brush definieren mit Zickzackmuster
        var hatchBrush = new HatchBrush(HatchStyle.ZigZag,
            //Zeichnen des Rechtecks
            graphics.FillRectangle(hatchBrush, new Rectangle(30,30,100,100));
    }
}
```

Listing B.28: Zeichnen eines Rechtecks mithilfe eines HatchBrushes

Das Ergebnis dieser Pinselvariation sieht so aus:

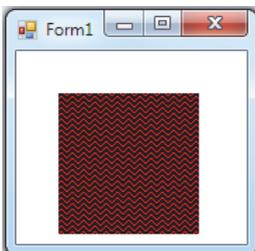


Abb. B.25: Zeichnen eines Rechtecks mithilfe eines HatchBrushes

B.8 Zusammenfassung

In diesem Kapitel wurde die Grafikprogrammierung mit GDI+ behandelt. Sie haben gelernt, wie sich das Koordinatensystem von GDI+ von dem Ihnen bekannten System unterscheidet. Weiterhin haben Sie die Klasse **Graphics** und deren zahlreiche Methoden zum Zeichnen von Objekten wie Linien, Rechtecken und Kreisen kennengelernt und erfahren, wie man mithilfe von Pfaden sogar komplexere Formen und Figuren erzeugen kann. Darüber hinaus haben wurde auch die Manipulation von Bildern mithilfe der Klassen **Image** und **Bitmap** und dem Steuerelement **PictureBox** besprochen. Zu guter Letzt sind Sie nun mit dem Farbsystem und der Ausgabe von Text als Grafik vertraut.

Stichwortverzeichnis

A

- Anchor 50
- Antialiasing 77
- Application 7
 - DoEvents 10
 - EnableVisualStyles 6, 10
 - Exit 9
 - Restart 9
 - Run 7, 9
 - SetCompatibleTextRenderingDefault 6
- ARGB siehe RGB

B

- Bezierkurve
 - DrawBezier 84
- Bézierkurve 84
- Bitmap
 - GetPixel 95
 - SetPixel 95
- Brush 102
 - HatchBrush 105
 - LinearGradientBrush 102
 - PathGradientBrush 104
 - SolidBrush 102
 - TextureBrush 103
- Button 24
 - FlatStyle 26

C

- CheckBox 26
 - Appearance 27
 - Checked 27
- CheckedListBox 28
 - Items 28
- Color 98
 - FromArgb 98
 - FromName 98
- ColorDialog 57
- ComboBox 30
- ContextMenuStrip 49

D

- DateTimePicker 31
- DialogResult 18

- Dock 50
- DrawArc 87
- DrawEllipse 85
- DrawLine 77
- DrawPie 86
- DrawRectangle 81

E

- Eigenschaftenfenster 12
- Ereignisse 21

F

- FlowLayoutPanel 52
 - WrapContents 52
- FolderBrowserDialog 59
- Font 97
 - FontStyle 97
- FontDialog 58
 - Font 59
- Form 10
 - Controls 23
 - FormBorderStyle 16
 - Name 15
 - Show 18
 - ShowDialog 18
 - Text 15
 - WindowState 15

G

- GDI 71
 - Graphics 71
 - Graphics Design Interface 71
 - System.Drawing 72
- GDI+ siehe GDI
- Gerätekoordinaten
 - Dpi 73
- Graphics 73
 - Paint 73
- GroupBox 52

I

- Image 91
 - Bitmap 40, 91
 - System.Drawing.Imaging 91

K

- Koordinatensystem 72
 - Gerätekoordinaten 73
 - globale Koordinaten 73
 - Seitenkoordinaten 73

L

- Label 32
 - LinkLabel 33
- LinearGradientBrush
 - LinearGradientMode 102
- ListBox 34
 - ListView 36
- ListView
 - BeginUpdate 37
 - EndUpdate 37
 - ListViewItem 36
 - SubItems 36

M

- MaskedTextBox
 - Mask 37
- MenuStrip 47
 - ToolStripMenuItem 47
- Modales Fenster 18
- MonthCalendar 39

N

- Nichtmodale Fenster 18
- NumericUpDown 39
 - DomainUpDown 40
 - Maximum 39
 - Minimum 39

P

- PageSetupDialog 63
- Panel 53
- Pen 99
- Pfad 88
 - DrawPath 88
 - GraphicsPath 88
- PictureBox 40, 91
 - Image 40
- Polygon 79
 - DrawPolygon 79
- Polyline 78
 - DrawLines 78
- PrintDialog 62
- PrintDocument 60
 - Print 61
 - PrintPage 60

- PrintPreviewControl 65
- PrintPreviewDialog 64
- Program 6
 - Main 6
 - STAThread 6
- ProgressBar 41

R

- RadioButton siehe CheckBox
- Resources.resx 69
- RGB 98
- RichTextBox 43
- RotateTransform 76

S

- ScaleTransform 75
- Settings siehe Properties
- Smarttag 56
- Spline 82
 - DrawCurve 82
- SplitContainer 53
 - Splitter 53
- StatusStrip 48
- SystemColors 98

T

- TabControl 54
 - TabPage 55
- TableLayoutPanel 56
- Textausgabe 96
 - DrawString 96
 - MeasureString 96
- TextBox 42
 - MaskedTextBox 37
 - Text 43
- ToolStrip 48
- ToolStripContainer 49
 - ToolStripPanel 49
- TranslateTransform 74
- TreeNode-Editor 45
- TreeView 44
 - TreeNode 44, 46

U

- UserControl 66

W

- Windows Forms 5
 - Form 7
 - InitializeComponent 7